8

LEVEL II

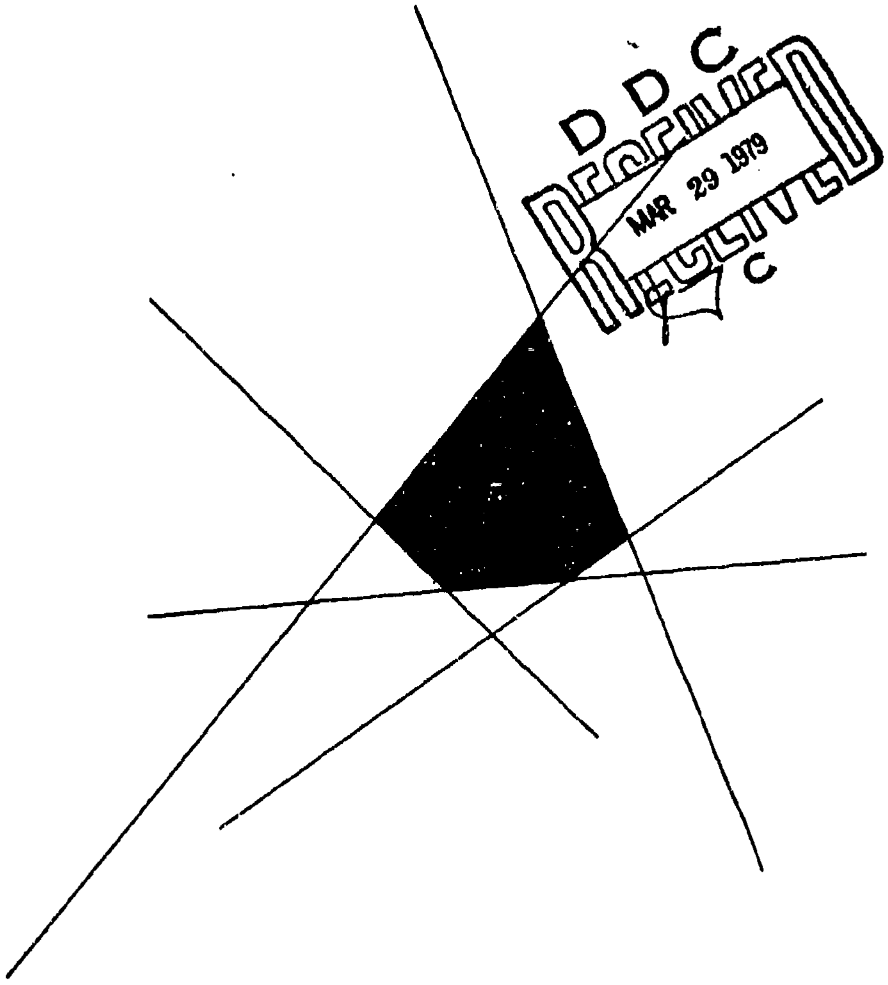
(12)

ORC 78-14
AUGUST 1978

COMPUTER-AIDED FAULT TREE ANALYSIS

by

RANDALL R. WILLIE



OPERATIONS
RESEARCH
CENTER

UNIVERSITY OF CALIFORNIA • BERKELEY

COMPUTER-AIDED FAULT TREE ANALYSIS[†]

by

Randall R. Willie
Operations Research Center
University of California, Berkeley

AUGUST 1978 ORC 78-14

This research has been partially supported by the Office of Naval Research under Contract N00014-75-C-0781, the Air Force Office of Scientific Research (AFSC), USAF, under Grant AFOSR-77-3179 and the Lawrence Livermore Laboratory under Purchase Order No. 7800103 with the University of California. Reproduction in whole or in part is permitted for any purpose of the United States Government.

[†]The computer program FTAP is available from NISEE/Computer Applications, Davis Hall, University of California, Berkeley, California 94720, (415) 642-5113.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) ORC-78-14	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) COMPUTER-AIDED FAULT TREE ANALYSIS,		5. TYPE OF REPORT & PERIOD COVERED (9) Research Report,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Randall R. Willie		8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0781, AFOSR-77-3179
9. PERFORMING ORGANIZATION NAME AND ADDRESS Operations Research Center University of California Berkeley, California 94720		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 042 238
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Department of the Navy Arlington, Virginia 22217		12. REPORT DATE (11) Aug 1978
		13. NUMBER OF PAGES 103
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 104p.		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Also supported by the Air Force Office of Scientific Research (AFSC), USAF, under Grant AFOSR-77-3179 and the Lawrence Livermore Laboratory under Purchase Order No. 7800103.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Fault Tree
Logic Tree
Minimal Cut Set
Prime Implicant

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(SEE ABSTRACT)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

270 750

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ACKNOWLEDGMENT

I am deeply indebted to Professor Richard E. Barlow of University of California, Berkeley for his guidance and encouragement in the preparation of this report and the associated computer program. I am also grateful for his cheerful patience as I consistently underestimated the remaining time to complete this project.

Dr. Howard Lambert of Tera Corporation (formerly of Lawrence Livermore Laboratories) deserves special thanks for providing a number of large fault trees to test the algorithms and program logic. Finally, I am grateful to Dr. Richard Worrell of Sandia Laboratories for a stimulating discussion of fault tree methods during his visit to Berkeley in November, 1977.

ABSTRACT

Part I of this report discusses a computer-oriented methodology for deriving minimal cut and path set families associated with arbitrary fault trees. Part II describes the use of the Fault Tree Analysis Program (FTAP), an extensive FORTRAN computer package that implements the Part I methodology. An input fault tree to FTAP may specify the system state as any logical function of subsystem or component state variables or complements of these variables. When fault tree logical relations involve complements of state variables, the analyst may instruct FTAP to produce a family of *prime implicants*, a generalization of the minimal cut set concept. FTAP can also identify certain subsystems associated with the tree as system modules and provide a collection of minimal cut set families that essentially expresses the state of the system as a function of these module state variables. Another FTAP feature allows a subfamily to be obtained when the family of minimal cut sets or prime implicants is too large to be found in its entirety; this subfamily consists only of sets that are "interesting" to the analyst in a special sense.

TABLE OF CONTENTS

INTRODUCTION

The analyst who seeks to determine reliability characteristics of a complex system, such as a nuclear reactor, in terms of the reliability characteristics of its subsystems and components confronts a number of difficult tasks. One task involves identification either implicitly or explicitly, of logical modes of system success or failure, that is, various distinct combinations of subsystems whose mutual success or failure implies success or failure of the entire system. Minimal cut set and path set families, tools familiar to reliability analysts for some time, provide an explicit representation of these modes. These families are useful not only for evaluating reliability characteristics of a system but also as a design tool to guide system modifications for enhancing reliability.

A widely used concept in reliability analysis of complex systems is that of a fault tree. Fault tree methods are based on the observation that the system state, either working or failed, can usually be expressed as a Boolean relation between states of several large, readily identifiable subsystems. The state of each subsystem in turn depends on states of simpler subsystems and components which compose it, so the state of the system itself is determined by a hierarchy of logical relationships between states of subsystems. A fault tree is a graphical representation of these relationships. At the lowest level of the hierarchy are subsystems whose success or failure dependence is not further described. If reliability information is available for these lowest level subsystems, then it may be possible to use this information to deduce reliability characteristics of the system itself.

An analyst who prepares a system fault tree often does so with the intention of utilizing it to obtain certain minimal cut (or path) set families in terms of these lowest level subsystems and components. Part I of this discussion outlines a computer-oriented methodology for deriving such families for an arbitrary fault tree. Part II describes the use of the Fault Tree Analysis Program (FTAP), an extensive computer package, written mostly in FORTRAN, which implements the Part I methodology.

FTAP has a number of useful features that make it well-suited to nearly all fault tree applications. An input fault tree to this program may specify the system state as any logical function of subsystem or component state variables or complements of these variables; thus, for instance, *exclusive - or* type relations may be formed. When fault tree logical relations involve complements of state variables, the concept of a minimal cut set family is no longer particularly useful, so in this case the analyst may instruct FTAP to produce a family of *prime implicants*, a generalization of the minimal cut set concept. The program offers the flexibility of several distinct methods of generating cut set families, and these methods may differ considerably in efficiency, depending on the particular tree analyzed. FTAP can also identify certain subsystems as system modules and provide a collection of minimal cut set families that essentially expresses the state of the system as a function of these module state variables. This collection is a compact way of representing the same information as contained in the system minimal cut set family in terms of lowest level subsystems and components. Another feature allows a useful subfamily to be obtained when a family of minimal cut sets or prime implicants is too large to be found in its entirety; this subfamily may consist of only sets not containing more than some fixed number of elements or only sets that are "interesting" to the analyst in a special sense. Finally, the analyst can modify the input fault tree in various ways by declaring state variables identically true or false.

A number of computer programs are currently available for obtaining minimal cut set families from fault trees, and some of these programs are mentioned in the discussion of Part I. One very capable package that deserves special mention is the SETS program developed by Dr. Richard Worrell of Sandia Laboratories [18]. In addition to fault tree analysis, SETS manipulates arbitrary Boolean expressions. For fault tree work, several features of FTAP and SETS are similar, and both programs have been used with good results during the past year in nuclear reactor safety studies conducted by Dr. Howard Lambert of the Lawrence Livermore Laboratories.

PART I

METHODS FOR COMPUTER-AIDED FAULT TREE ANALYSIS

The first two sections below essentially provide notation and background information for the procedures presented in Sections I.3 and I.4. The notation introduced in Section I.1 has been chosen both to reflect the computer implementation of these procedures and to relate their various operations to manipulation of Boolean expressions. In Section I.2, fault trees and implicant families are formally defined, and two quite well-known fault tree algorithms, MOCUS and MICSUP, are reviewed.

The reader who is primarily interested in using FTAP should look over Section I.1 and Subsections I.2.1, I.2.2, I.3.1, I.3.2, and I.4.4 before skipping to Part II.

I.1 Boolean Expressions

The reader is assumed to be familiar with the rudiments of Boolean algebra; a reference such as [16], for instance, is more than adequate as background. Let $x_1, \ldots, x_q$ be Boolean variables independently taking on values of 0 or 1, and let $\underline{x} \equiv (x_1, \ldots, x_q)$ be a vector of 0's and 1's representing an arbitrary choice of these values. We denote complementation by negation of subscripts: For any u in the set $U = [1, \ldots, q]$, $\bar{x}_u$ $(\equiv 1 - x_u)$ is written as x_{-u}. The index set for complements is $-U \equiv [-1, \ldots, -q]$, and $(u,-u)$ is a complementary pair of indices.

Expressions may be formed using $x_1, \ldots, x_q$, $x_{-1}, \ldots, x_{-q}$ and the ordinary Boolean relations of product and sum. An arbitrary non-empty family $\mathcal{I}$ of subsets of $U \cup (-U)$ (not necessarily distinct) is identified with the Boolean sum-of-products expression

$$\sum_{I \in \mathcal{I}} \prod_{i \in I} x_i \ .$$

The notation $/\mathcal{I}/\underline{x}$ denotes the value of this expression for a given vector $\underline{x}$ of 0's and 1's , that is,

$$/\mathcal{I}/\underline{x} \equiv \max_{I \in \mathcal{I}} \left(\min_{i \in I} x_i \right) = \sum_{I \in \mathcal{I}} \prod_{i \in I} x_i \ .$$

$/\mathcal{I}/$ may then be taken as a Boolean function mapping each vertex of the q-dimensional unit cube into 0 or 1 . Given nonempty families $\mathcal{I}$, $\mathcal{J}$, and $\mathcal{K}$ of subsets of $U \cup (-U)$, $/\mathcal{I}/ \equiv /\mathcal{J}/$ means that for all $\underline{x}$ $/\mathcal{I}/\underline{x} = /\mathcal{J}/\underline{x}$. Similarly, if for all $\underline{x}$ $/\mathcal{I}/\underline{x} = /\mathcal{J}/\underline{x} + /\mathcal{K}/\underline{x}$ $(/\mathcal{I}/\underline{x} = /\mathcal{J}/\underline{x} \cdot /\mathcal{K}/\underline{x})$, write $/\mathcal{I}/ \equiv /\mathcal{J}/ + /\mathcal{K}/$ $(/\mathcal{I}/ \equiv /\mathcal{J}/ \cdot /\mathcal{K}/)$. For the null family $(\emptyset)$ we define $/\emptyset/ \equiv 0$; although for the family containing only the empty set $([\emptyset])$, $/[\emptyset]/$ is left undefined.

The union of families $\mathcal{I}$ and $\mathcal{J}$ clearly has the property

$$/\mathcal{I} \cup \mathcal{J}/ \equiv /\mathcal{I}/ + /\mathcal{J}/ \ .$$

Now suppose $U = \{1,2,3\}$ and $\mathcal{I}_1 = [\{2,3\}]$ $\mathcal{I}_2 = [\{1,2,3\}]$ and $\mathcal{I}_3 = [\{-1,-3,3\}]$. For any $\underline{x} \equiv (x_1,x_2,x_3)$, $/\mathcal{I}_3/\underline{x} = x_{-1}x_{-3}x_3 = 0$, so $/\mathcal{I}_1 \cup \mathcal{I}_2 \cup \mathcal{I}_3/ \equiv /\mathcal{I}_1 \cup \mathcal{I}_2/$, and $\mathcal{I}_3$ need not be considered further. Thus to simplify the discussion, *it is assumed that no set of a family contains a complementary pair*; whenever a new family is constructed, any sets containing complementary pairs are simply eliminated.

In the example above it is also true that for all $\underline{x}$, $/I_1/\underline{x} = /I_2 \cup I_3/\underline{x}$, since $\{2,3\} \subset \{1,2,3\}$, and thus $/I_1/\underline{x} = 1$ whenever $/I_2/\underline{x} = 1$. A family is said to be *minimal* if all sets are distinct and for any two sets of the family, neither is a subset of the other. For any family I , let $m[I]$ (the "minimization" of I) be the minimal family obtained by eliminating duplicate sets and those which contain another set of I . For instance, $m[[\{2,3\},\{1,2,3\}]] = [\{2,3\}]$. Of course, for any I , $/m[I]/ \equiv /I/$.

Next, the *product* family $I \times J$ of two families $I \neq \emptyset$ and $J \neq \emptyset$ is defined by $[I \cup J \mid I \in I , J \in J]$; that is, $I \times J$ consists of all possible sets that may be formed by taking the union of a set from I and a set from J , excluding unions which contain complementary pairs. The product is assumed to be empty if either I or J is empty. Evidently, $/I \times J/ \equiv /I/ \cdot /J/$ since for all $\underline{x}$,

$$/I/\underline{x} \cdot /J/\underline{x} = \left(\sum_{I\in I} \prod_{i\in I} x_i\right)\left(\sum_{J\in J} \prod_{j\in J} x_j\right)$$

$$= \sum_{K\in I\times J} \prod_{k\in K} x_k .$$

We will need one additional concept. Given a nonempty family I of subsets of $U \cup (-U)$, the *dual family* of I , denoted by $d[I]$, consists of all distinct sets J such that $J \cap I \neq 0$ for each $I \in I$ and no subset of J has this property. By definition, $d[I]$ is always minimal, and though I may not be minimal, it is not difficult to see $d[I] = d[m[I]]$. In general, $d[d[I]] \neq I$, though there is one important case in which equality holds: If I is a minimal family of subsets of U (rather than $U \cup (-U)$) then I is called a *clutter*, and $d[I]$ is then known as the *blocker* of I , usually written as $b[\mathcal{I}]$. It can be shown [5] that $b[b[\mathcal{I}]] = \mathcal{I}$. If $\mathcal{I}$ consists of subsets of $U \cup (-U)$, however, then $d[\mathcal{I}]$ may be empty; for instance, let $\mathcal{I} = [\{-1\},\{1\}]$.

The dual family is useful because it allows us to relate an expression in product-of-sums form to a sum-of-products form. Some thought indicates that for all $\underline{x}$,

$$\prod_{I\in\mathcal{I}} \sum_{i\in I} x_i = \sum_{I\in d[\mathcal{I}]} \prod_{i\in I} x_i$$

and

$$\sum_{I\in\mathcal{I}} \prod_{i\in I} x_i = \prod_{I\in d[\mathcal{I}]} \sum_{i\in I} x_i .$$

The following simple propositions will be useful later on:

<u>Proposition I.1.1</u>:

If $\mathcal{I} \neq \emptyset$ then for all $\underline{x}$, $1 - /d[\mathcal{I}]/(\underline{1} - \underline{x}) = /\mathcal{I}/\underline{x}$, where $\underline{1} - \underline{x}$ is the vector $(1 - x_1, \ldots, 1 - x_n)$.

This is true because De Morgan's Law gives

$$\sum_{I\in\mathcal{I}} \prod_{i\in I} x_i = 1 - \prod_{I\in\mathcal{I}} \sum_{i\in I} (1 - x_i) ,$$

and the value of the expression on the right equals

$$1 - \sum_{I\in d[\mathcal{I}]} \prod_{i\in I} (1 - x_i) ,$$

which is $1 - /d[\mathcal{I}]/(\underline{1} - \underline{x})$.

Proposition I.1.2:

If $I \neq \emptyset$ and $J \neq \emptyset$, then

$$d[I \cup J] = m[d[I] \times d[J]] \ .$$

It is easy to see that $d[I \cup J] \subseteq d[I] \times d[J]$, and each set of $d[I] \times d[J]$ equals or contains a set of $d[I \cup J]$, so the above proposition follows. A corollary is:

Proposition I.1.3:

For $I \neq \emptyset$ and $J \neq \emptyset$ if $d[d[I]] = I$ and $d[d[J]] = J$ then

$$d[d[I] \cup d[J]] = m[I \times J] \ .$$

I.2 Fault Tree Fundamentals

Some of the terminology and notation of Subsection I.2.1 is not in standard use for fault tree work, partly because fault trees are traditionally defined in a manner that does not permit system failure to depend on complements of Boolean state variables for subsystems or components. Subsection I.2.2, which presents the MOCUS and MICSUP methods in the context of this notation and terminology, serves as a useful introduction to the algorithms of Section I.4. The final subsection, I.2.3, formalizes the idea of a subfamily of "interesting" cut sets.

I.2.1 Fault Tree Definitions

Formally, a fault tree is an acyclic directed graph (U,A) , where U is the set of nodes and A is the set of arcs. Any pair of nodes may be joined by at most a single arc, which may be either a *regular* arc or a *complementing* arc. Nodes having no entering arcs we call *basic* nodes, and those having one or more entering arcs are called *gate* nodes. Those which have no leaving arcs are *top* nodes; a fault tree usually has only a single top node. The tree is drawn with arc paths directed upward from basic nodes and terminating at the top node. Nodes are numbered by consecutive positive integers, with gates numbered first. Also, associated with each gate is a *logic indicator*, a positive integer ℓ that may take on any value between 1 and the number of entering arcs for that gate.

Figure 1 presents a typical fault tree to illustrate the above terminology. Basic nodes are denoted by circles and gate nodes by rectangles with node 1 as the top. All arcs are regular with the exception of the complementing arc joining nodes 6 and 4, and this arc is distinguished by the symbol "~." The logic indicator for each gate node appears in the lower half of the rectangle; ℓ_1 , ℓ_3 , ℓ_7 and ℓ_8 are all 1 , and ℓ_2 , ℓ_4 , ℓ_5 , and ℓ_6 are equal to 2 .

We say that node v is a *subnode* of node u if there is an arc path directed upward from v to u , and v is an *immediate* subnode of u if there is a single upward arc from v to u . Nodes 7, 8, 12, 13, and 14, for instance, are subnodes of node 5; whereas, 7 and 8 are the immediate subnodes of 5. When v is a subnode (immediate subnode) of u , u is sometimes referred to as a *supernode* *(immediate supernode)* of v .

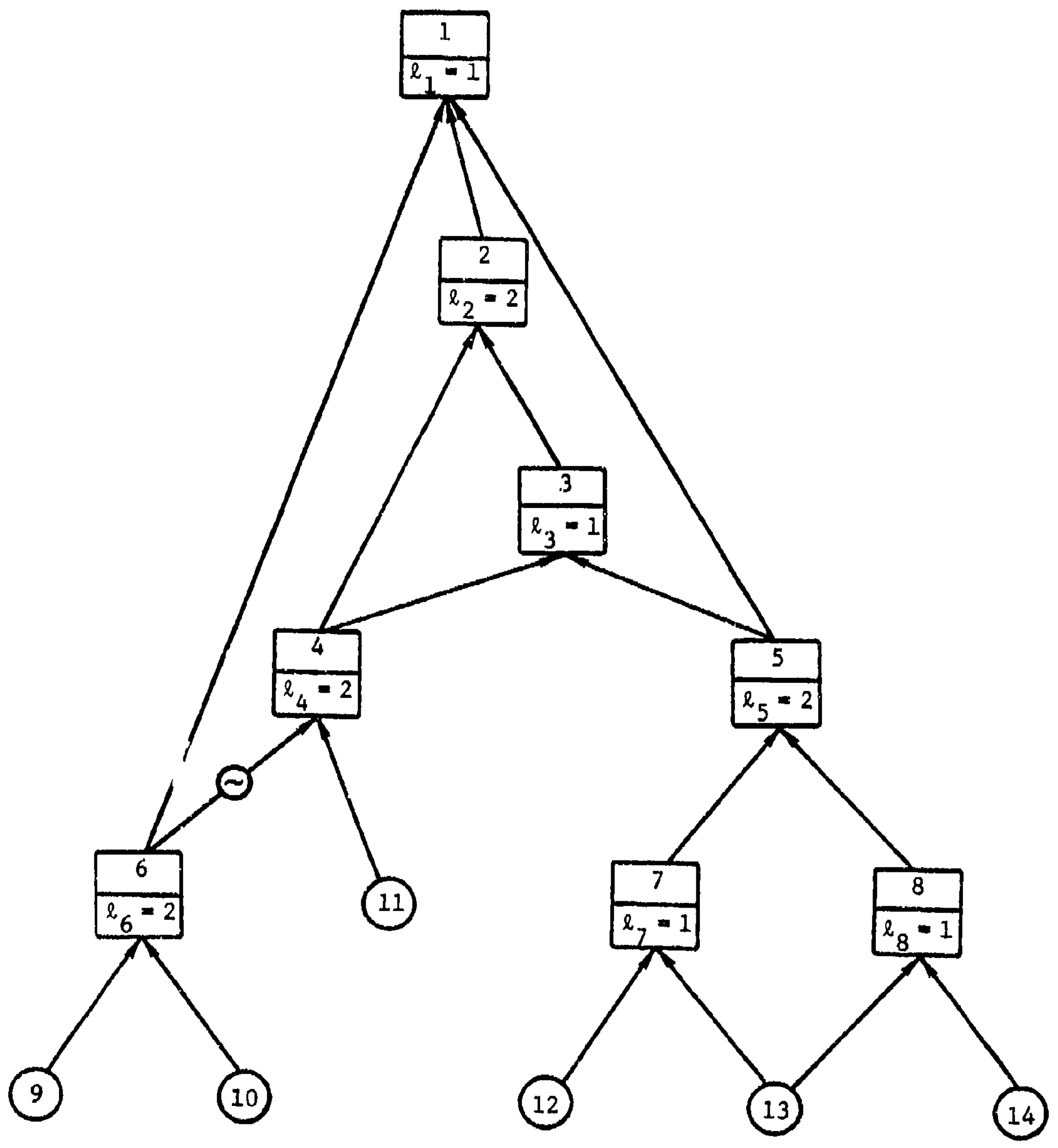

1
$\ell_1 = 1$
2
$\ell_2 = 2$
3
$\ell_3 = 1$
4
$\ell_4 = 2$
5
$\ell_5 = 2$
6
$\ell_6 = 2$
7
$\ell_7 = 1$
8
$\ell_8 = 1$
~
9
10
11
12
13
14

FIGURE 1

Finally, given a set of nodes $V \subseteq U$, a *downward order* on V is any complete ordering $(\overset{d}{\succ})$ of nodes of V such that for any $v , w \in V$, $v \overset{d}{\succ} w$ implies v is not a subnode of w . On the other hand, if $w \overset{u}{\succ} v$ implies v is not a subnode of w , then the order $(\overset{u}{\succ})$ is an *upward order*. Thus, for $V = \{2,3,4,5\}$, $2 \overset{d}{\succ} 3 \overset{d}{\succ} 5 \overset{d}{\succ} 4$ is a downward order and $5 \overset{u}{\succ} 4 \overset{u}{\succ} 3 \overset{u}{\succ} 2$ is an upward order.

A fault tree is a convenient representation of a system of Boolean expressions. Let the set of nodes be $U = \{1, \ldots, p - 1,p,p + 1, \ldots, q\}$ where $G = \{1, \ldots, p\}$ are gate nodes and $B = \{p + 1, \ldots, q\}$ are basic nodes. With the uth node we associate the Boolean variable x_u . If u is a basic node, then x_u may take on the value 0 or 1 , independently of the values of other node variables; thus, $\underline{b} \equiv (x_{p+1}, \ldots, x_q)$ is an arbitrary vector of 0's and 1's whose elements are a particular choice of these values. On the other hand, if u is a gate node, then the value of x_u ultimately depends on values of the independent basic node variables; that is, x_u is a Boolean function of the vector $\underline{b}$. Gate variable values are determined by the following scheme: Let D_u be a set of integers representing the immediate subnodes of u . If node v is joined to node u by a regular arc then $v \in D_u$; if v is joined to u by a complementing arc then $-v \in D_u$. Note that since only a single arc can join any two events in the tree, D_u contains no complementary pair $(v,-v)$. The *node definition family* $\mathcal{D}(\ell_u,D_u)$ is a family of subsets of $U \cup (-U)$ that consists of all possible sets of size ℓ_u that may be formed from the elements of D_u , where ℓ_u is the logic indicator for node u . The value of x_u is determined by

$$x_u = \sum_{I \in \mathcal{D}(\ell_u, D_u)} \prod_{i \in I} x_i \ .$$

Each integer $i \in I$ may thus be positive or negative, with $x_{-i} \equiv (1 - x_i)$. An informal statement of this relation is that the u^th gate node is "true" iff ℓ_u or more of its inputs are "true." The logic indicator satisfies $1 \leq \ell_u \leq \#D_u$, where $\#D_u$ represents the number of elements in D_u . The value $\ell_u = 1$ corresponds to the "OR" relation between immediate subnode variables (or their complements); that is,

$$x_u = \sum_{i \in D_u} x_i \ ;$$

whereas, $\ell_u = \#D_u$ represents the "AND" relation,

$$x_u = \prod_{i \in D_u} x_i \ .$$

If we apply De Morgan's Law to the general expression for x_u , u a gate node, a similar expression may be obtained for x_{-u} . Let $D_{-u} = -D_u \equiv \{-i \mid i \in D_u\}$. Then

$$x_{-u} = \sum_{I \in \mathcal{D}(\#D_u - \ell_u + 1, D_{-u})} \prod_{i \in I} x_i \ .$$

Since variables x_u and x_{-u} are each associated with node u , it is convenient to call both indices u and $-u$ events; $-u$ is the *complementary* event for node u .

Let $\underline{x} = (x_1, \ldots, x_{p-1}, x_p, x_{p+1}, \ldots, x_q)$ be a vector of 0's and 1's. Using the notation developed in Section I.1, $\underline{x}$ will be said to be *consistent with the fault tree* if for all gate nodes $u \in G$, $x_u = /\mathcal{D}(\ell_u, D_u)/\underline{x}$. Thus the set of all vectors $\underline{x}$ consistent with the fault tree is a subset of vertices of the q-dimensional unit cube that represents all logically possible combinations of states of the system and its subsystems and components. If $\underline{x}$ and $\underline{x}'$ are both consistent with the fault tree and have the same values for basic node variables (i.e., $x_{p+1} = x'_{p+1}, \ldots, x_q = x'_q$), then it will be the case that $\underline{x} = \underline{x}'$. So we might write a consistent vector as $\underline{x}(\underline{b}) = (x_1(\underline{b}), \ldots, x_{p-1}(\underline{b}), x_p(\underline{b}), \underline{b})$ for some vector $\underline{b} = (x_{p+1}, \ldots, x_q)$ of values for basic node variables.

A subset F of $U \cup (-U)$ is called an *implicant set* (or just *implicant*) *for event* i if $x_i = 1$ for every vector $\underline{x}$ consistent with the fault tree such that $/[F]/\underline{x} = 1$. A family $\mathcal{F}$ of subsets of $U \cup (-U)$ is termed an *implicant family for event* i if for all consistent $\underline{x}$, $x_i = /\mathcal{F}/\underline{x}$. Thus an implicant family for event i is a particular collection of implicants for event i. Naturally, an implicant family $\mathcal{F}$ for some event is minimal when $m[\mathcal{F}] = \mathcal{F}$. As an example, some of the minimal implicant families for event 1 of the tree of Figure 1 are $[\{2\},\{5\},\{6\}]$, $[\{2\},\{7,8\},\{9,10\}]$, and $[\{3,4\},\{5\},\{6\}]$.

Some additional definitions are useful in dealing with fault trees involving complementing arcs. Again, let $\underline{x} = (x_1, \ldots, x_q)$ be any vector of 0's and 1's (not necessarily consistent with the fault tree), and suppose ϕ is a Boolean function mapping each such $\underline{x}$ into 0 or 1. A subset P of $U \cup (-U)$ is a *prime implicant* *of* $\underline{\phi}$ if P implies ϕ (i.e., $\phi(\underline{x}) = 1$ for all $\underline{x}$ such that $/[P]/\underline{x} = 1$) and no proper subset of P implies ϕ. Also, a family $\mathcal{P}$ of distinct subsets of $U \cup (-U)$ is a *prime implicant family* for ϕ if for all $\underline{x}$ $\phi(\underline{x}) = /\mathcal{P}/\underline{x}$ and each $P \in \mathcal{P}$ is a prime implicant for ϕ. The concepts of prime implicants and prime implicant families have been widely applied in the fields of switching theory and logic, and the definitions given here are standard in most introductory textbooks devoted to these fields. If $\mathcal{P}_1$ and $\mathcal{P}_2$ are both prime implicant families for ϕ, then $\mathcal{P}_1 = \mathcal{P}_2$, so a prime implicant family is unique.

Specializing the idea of a prime implicant family to our purposes here, we will call a family $\mathcal{P}$ of subsets of $U \cup (-U)$ a *prime implicant family for event* i if $\mathcal{P}$ is an implicant family for event i and each $P \in \mathcal{P}$ is a prime implicant of the Boolean function $/\mathcal{P}/$. Note that if $\mathcal{F}$ is an implicant family for event i, the situation $/[F]/\underline{x} = 1$ for some $F \in \mathcal{F}$ requires that $x_i = 1$ only if $\underline{x}$ is consistent with the fault tree; however, whether F is a prime implicant of $/\mathcal{F}/$ depends on all $\underline{x}$, not just vectors consistent with the fault tree. Thus with this definition, two prime implicant families $\mathcal{P}_1$ and $\mathcal{P}_2$ for event i need not be the same, since $/\mathcal{P}_1/\underline{x} = /\mathcal{P}_2/\underline{x}$ need only hold for consistent $\underline{x}$. But if $\mathcal{P}_1$ and $\mathcal{P}_2$ are composed only of sets of basic events, it will be true that $\mathcal{P}_1 = \mathcal{P}_2$. In fact, in sections which follow, we will not be concerned with whether an implicant family $\mathcal{F}$ for event i consists of prime implicants for $/\mathcal{F}/$ unless $\mathcal{F}$ consists only of subsets of basic events or only of *largest simple modules for event* i, which will be introduced in Section I.3.

As an example, the family $F = [\{9,10\},\{12,14\},\{13\},\{-9,11\},\{-10,11\}]$ is an implicant family for event 1 of the tree of Figure 1, and F is in terms of basic events; but F is not a prime implicant family for event 1. On the other hand, $P = [\{9,10\},\{12,14\},\{13\},\{11\}]$ is a prime implicant family for event 1 and it may be verified that $/P/ \equiv /F/$.

The fault tree algorithms MOCUS and MICSUP discussed in the following subsection obtain, for selected gate events of the tree, minimal implicant families in terms of basic events, that is, families of subsets of $B \cup (-B)$ ·· family F obtained by one of these methods will not in general t. a prime implicant family unless F consists only of subsets of B (or only of subsets of $-B$). When F consists only of subsets of B , F is usually called a *minimal cut set family*. The dual family in this case is the family of minimal *path* sets.

In utilizing a fault tree to obtain information on the reliability of a system, it is necessary to have on hand estimates of the probability of failure of components or subsystems associated with the basic events; the availability of these estimates determines the extent to which subsystems are broken down into further subsystems. Once minimal implicant family in terms of basic events has been obtained for the subtree top event, bounds on system reliability can often be found, as well as a number of measures of the contribution of basic event components and subsystems to reliable system operation. The use of cut sets in reliability evaluation is discussed by Barlow and Proschan [2] and Lambert [10].

I.2.2 The MOCUS and MICSUP Methods

Under the name MOCUS (Method of Obtaining Cut Sets), Vesely and Fussell [6] suggested one of the first methods for finding a minimal implicant family in terms of basic events for the top node event (or any other gate event) of the tree. MOCUS was originally proposed for fault trees that do not include complementing arcs, but the method remains essentially unchanged if complementing arcs are present. A computer program which implements MOCUS is the subject of Reference [7].

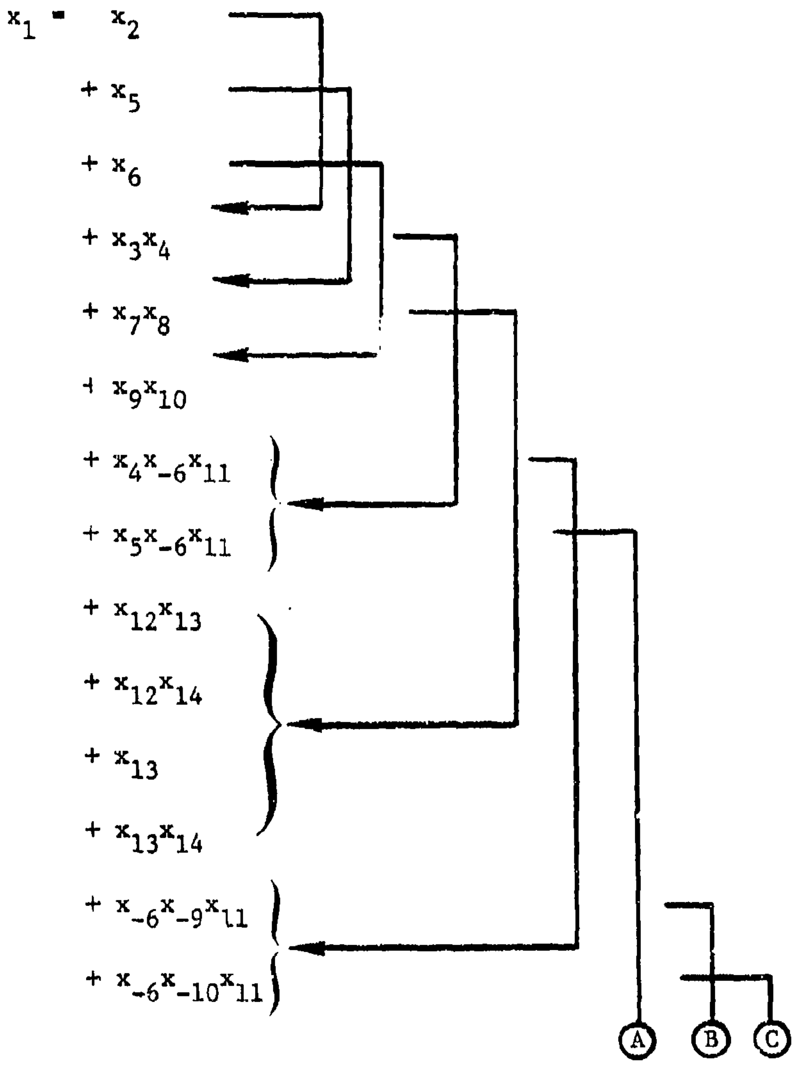
For the top node event, say event 1, the procedure begins with the definition family $\mathcal{D}(\ell_1, D_1)$ and generates a succession of implicant families for x_1 by continually replacing implicants involving gate events with events nearer the bottom of the tree. The essence of the method is summarized by the following steps:

```
0. $\mathcal{H} \leftarrow \mathcal{D}(\ell_1, D_1)$ .
1. If all sets $H \in \mathcal{H}$ have been considered in this step, go to
   4. Otherwise, select an $H \in \mathcal{H}$ not previously considered.
2. If all $e \in H$ are basic events, go to 1. Otherwise for
   each $e \in H$ that is a gate event $\mathcal{J}_e \leftarrow \mathcal{D}(\ell_e, D_e)$ , and for
   each $e \in H$ that is basic event $\mathcal{J}_e \leftarrow [\{e\}]$ .
3. $\mathcal{H} \leftarrow [\mathcal{H} - [H]] \cup \left[ \underset{e \in H}{X} \mathcal{J}_e \right]$ .
4. $\mathcal{I}_1 \leftarrow m[\mathcal{H}]$ .
```

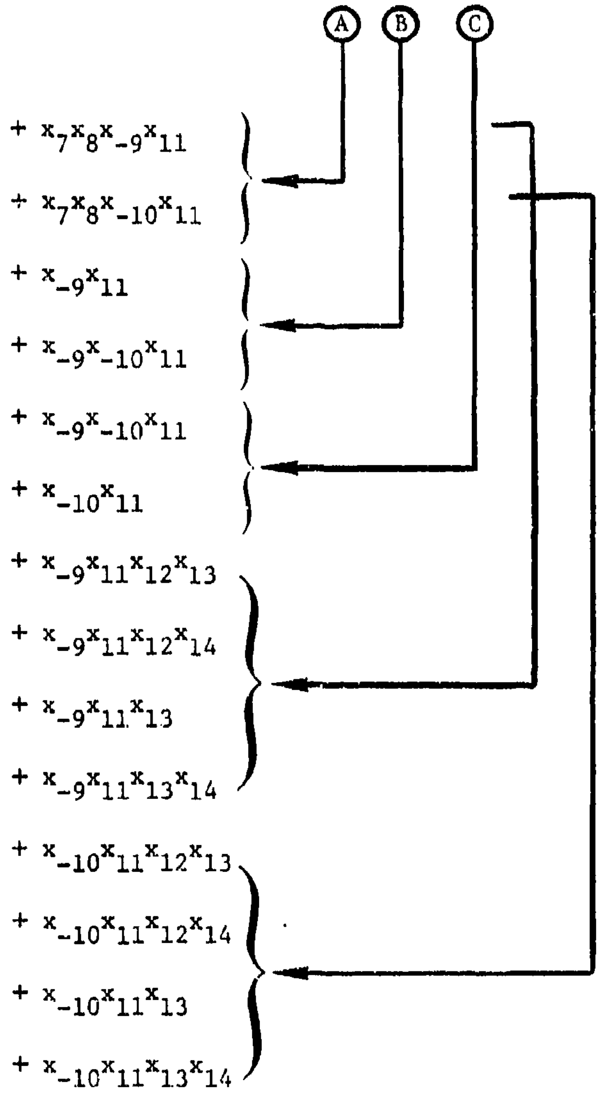
(The notation "symbol ← formula" is well-established and means that after the operations indicated by the formula on the right have been performed, the resulting object, whether it be a family, set, or quantity, is to be represented by the symbol on the left.) It is readily verified that $\mathcal{I}_1$ will be an implicant family for event 1.

The minimization task of Step 4 essentially consists of comparing each set $H \varepsilon \mathcal{H}$ with sets which precedes it in $\mathcal{H}$.

The algorithm as stated is suitable for computer implementation, but the main idea is best illustrated by deriving a Boolean sum-of-products expression in terms of basic event variables for event 1 of the tree of Figure 1. This expression is derived by repeated substitution for gate event variables. Since the x_j are Boolean variables, note that the identity $x_j^2 \equiv x_j$ may be used to simplify a product and that products involving a complementary pair of variables $(x_j, \bar{x}_j)$ may be discarded:



$$
\begin{aligned}
x_1 = \; & x_2 \\
& + x_5 \\
& + x_6 \\
& + x_3x_4 \\
& + x_7x_8 \\
& + x_9x_{10} \\
& + x_4x_{-6}x_{11} \\
& + x_5x_{-6}x_{11} \\
& + x_{12}x_{13} \\
& + x_{12}x_{14} \\
& + x_{13} \\
& + x_{13}x_{14} \\
& + x_{-6}x_{-9}x_{11} \\
& + x_{-6}x_{-10}x_{11}
\end{aligned}
$$



$$+ x_7x_8x_{-9}x_{11}$$
$$+ x_7x_8x_{-10}x_{11}$$
$$+ x_{-9}x_{11}$$
$$+ x_{-9}x_{-10}x_{11}$$
$$+ x_{-9}x_{-10}x_{11}$$
$$+ x_{-10}x_{11}$$
$$+ x_{-9}x_{11}x_{12}x_{13}$$
$$+ x_{-9}x_{11}x_{12}x_{14}$$
$$+ x_{-9}x_{11}x_{13}$$
$$+ x_{-9}x_{11}x_{13}x_{14}$$
$$+ x_{-10}x_{11}x_{12}x_{13}$$
$$+ x_{-10}x_{11}x_{12}x_{14}$$
$$+ x_{-10}x_{11}x_{13}$$
$$+ x_{-10}x_{11}x_{13}x_{14}$$

The sum of remaining products corresponds to a nonminimal implicant family for event 1. Minimization of this family is equivalent to applying the Boolean absorption identity $(x_i + x_ix_j \equiv x_i)$ to pairs of products in the above sum, thus eliminating redundant products. The resulting "minimal" expression is

$$x_1 = x_9x_{10} + x_{12}x_{14} + x_{13} + x_{-9}x_{11} + x_{-10}x_{11} .$$

MICSUP (MInimal Cut Sets, UPward) is an alternative method of constructing basic event implicant families proposed by Chatterjee [3]. At least two computer codes utilizing this method are available [12], [13]. The technique is based on the observation that if minimal basic event families I_j are available for all immediate subevents $j \in D_i$ of a particular gate event i, then the minimal basic event family for i is simply

$$m\left[\bigcup_{I \in \mathcal{D}(\ell_i, D_i)} \underset{j \in I}{\mathrm{X}} I_j\right],$$

where $I_j \equiv [\{j\}]$ if j is a basic event. To find a basic event family for the top node event of a fault tree, say event 1, the procedure is as follows:

0. $F \leftarrow \{1\}$.
1. If all events $i \in F$ have been considered in this step, go to 3. Otherwise select $i \in F$ not previously considered.
2. $F \leftarrow F \cup \{j \in D_i$, j a gate event$\}$.
3. Consider successive elements of F in upward order (any ordering such that each event follows all of its subevents). For each $i \in F$ construct

$$I_i \leftarrow m\left[\bigcup_{I \in \mathcal{D}(\ell_i, D_i)} \underset{j \in I}{\mathrm{X}} I_j\right].$$

Steps 0, 1, and 2 serve only to avoid, if possible, finding I_u and I_{-u} for every gate node u of a fault tree containing complementing arcs; if the tree contains no complementing arcs, we may let $F = G$, the set of gate nodes, and just perform Step 3. Also, minimization may be postponed until I_1 is constructed if it is expected that the unminimized families for immediate subevents of I_1 will not contain a large number of nonminimal sets.

For the example tree of Figure 1, this simple method is illustrated with Boolean expressions. Evidently, the set F in its proper order is {8,7,6,-6,5,4,3,2,1} .

$$x_8 = x_{13} + x_{14}$$

$$x_7 = x_{12} + x_{13}$$

$$x_6 = x_9 x_{10}$$

$$x_{-6} = x_{-9} + x_{-10}$$

$$\begin{aligned} x_5 &= x_7 x_8 \\ &= (x_{12} + x_{13})(x_{13} + x_{14}) \\ &= x_{12}x_{13} + x_{12}x_{14} + x_{13} + x_{13}x_{14} \\ &= x_{12}x_{14} + x_{13} \end{aligned}$$

$$\begin{aligned} x_4 &= x_{-6} x_{11} \\ &= (x_{-9} + x_{-10})x_{11} \\ &= x_{-9}x_{11} + x_{-10}x_{11} \end{aligned}$$

$$\begin{aligned} x_3 &= x_4 + x_5 \\ &= (x_{-9}x_{11} + x_{-10}x_{11}) + (x_{12}x_{14} + x_{13}) \\ &= x_{-9}x_{11} + x_{-10}x_{11} + x_{12}x_{14} + x_{13} \end{aligned}$$

$$
\begin{aligned}
x_2 &= x_3x_4 \\
&= (x_{-9}x_{11} + x_{-10}x_{11} + x_{12}x_{14} + x_{13})(x_{-9}x_{11} + x_{-10}x_{11}) \\
&= x_{-9}x_{11} + x_{-9}x_{-10}x_{11} + x_{-9}x_{11}x_{12}x_{14} \\
&\quad + x_{-9}x_{11}x_{13} + x_{-9}x_{-10}x_{11} + x_{-10}x_{11} \\
&\quad + x_{-10}x_{11}x_{12}x_{14} + x_{-10}x_{11}x_{13} \\
&= x_{-9}x_{11} + x_{-10}x_{11} \\
x_1 &= x_2 + x_3 + x_6 \\
&= (x_{-9}x_{11} + x_{-10}x_{11}) + (x_{-9}x_{11} + x_{-10}x_{11} + x_{12}x_{14} + x_{13}) \\
&\quad + (x_9x_{10}) \\
&= x_{-9}x_{11} + x_{-10}x_{11} + x_{-9}x_{11} + x_{-10}x_{11} \\
&\quad + x_{12}x_{14} + x_{13} + x_9x_{10} \\
&= x_{-9}x_{11} + x_{-10}x_{11} + x_{12}x_{14} + x_{13} + x_9x_{10} \ .
\end{aligned}
$$

The MICSUP algorithm is superior to MOCUS in two cases: (1) when basic event families are desired for a number of intermediate gate events as well as the top event, and (2) when only sets not exceeding some given number of basic events are required in the top event family. The second case is most important in practice. Often the minimal top event family has many sets which contain a large number of basic events. If the fault tree is free of complementing arcs, each of these sets is associated with a mode of system failure due to failure of a large number of basic node components and subsystems; that the actual system will fail in this manner is highly unlikely. Thus implicants which exceed some given size are usually not of interest in the subsequent reliability analysis of the system. The convenience of the MICSUP method in this case is a consequence of the fact that sets generated for every gate event consist only of basic events, and any set that exceeds a given size may be immediately discarded.

This way of finding a subfamily I_i' of the complete minimal family I_i for event i does not, in general, yield meaningful results for fault trees involving complementing arcs, due to the fact that I_i is usually not a prime implicant family. However, as an illustration, suppose that only products of size 1 are required in the previous example. The expression following each colon (:) below results from discarding products containing more than a single variable.

$$x_8 = x_{13} + x_{14}$$

$$x_7 = x_{12} + x_{13}$$

$$x_6 = x_9 x_{10} \ : \ \emptyset$$

$$x_{-6} = x_{-9} + x_{-10}$$

$$\begin{aligned} x_5 &= x_7 x_8 \\ &= (x_{12} + x_{13})(x_{13} + x_{14}) \ : \ x_{13} \end{aligned}$$

$$\begin{aligned} x_4 &= x_{-6} x_{11} \\ &= (x_{-9} + x_{-10}) x_{11} \ : \ \emptyset \end{aligned}$$

$$\begin{aligned} x_3 &= x_4 + x_5 \\ &= (\emptyset) + (x_{13}) \ : \ x_{13} \end{aligned}$$

$$\begin{aligned} x_2 &= x_3 x_4 \\ &= (x_{13})(\emptyset) \ : \ \emptyset \end{aligned}$$

$$\begin{aligned} x_1 &= x_2 + x_5 + x_6 \\ &= (\emptyset) + (x_{13}) + (\emptyset) \ : \ x_{13} \ . \end{aligned}$$

(A null expression for x_i means that there is no basic event variable which by itself implies x_i .) In addition to x_{13} , x_{11} also implies x_1 , so the method has failed in this case to find all single variables that imply x_1 . What is really desired here is the subfamily of all prime implicants for event 1 which consist of only one basic event.

I.2.3 General Framework for Implicant Elimination

Implicant size is one criterion which may be used to determine a subfamily of "interesting" implicants when the complete minimal implicant family is too large to obtain. More generally, given any set of events E , an *importance criterion* for E assigns certain subsets of E to a class, called *important* sets, in such a manner that if I is important, all subsets of I (ignoring the null set) are also important. This definition just guarantees that if $\mathcal{I}'$ is a subfamily of all important sets of the family $\mathcal{I}$ then $m[\mathcal{I}'] \subseteq m[\mathcal{I}]$. Also, let f be a real-valued function whose domain consists of all subsets of E ; it is convenient to call f an *importance function* if for any real c either $[I \mid I \subseteq E\ ,\ f(I) \leq c]$ or $[I \mid I \subseteq E\ ,\ f(I) > c]$ is a class of important sets.

Suppose that a positive real value $\iota(k)$ is chosen for each event $k \in E$. If $f(I) \equiv \sum_{k \in I} \iota(k)$ for each $I \subseteq E$, then f is an importance function; if $\iota(k) = 1$ for all $k \in E$, then all sets not exceeding a critical size c are important. Many other importance functions can be constructed using the $\iota(\cdot)$ values, such as $\min_{k \in I} \iota(k)$ or, when all values are between 0 and 1 , $\prod_{k \in I} \iota(k)$.

For fault trees that do not contain complementing arcs, the MICSUP algorithm obviously lends itself well to construction of minimal subfamilies of all implicants that satisfy an importance criterion. For $E = B$, as in the case of elimination by size, implicants that are not important may be discarded whenever they appear.

I.3 Simple Modules

Deleting nonminimal implicants of a family is unquestionably the most time consuming task in MOCUS and MICSUP methods. Given an arbitrary implicant family $\mathcal{K}$ in any order, $m[\mathcal{K}]$ is obtained essentially by comparing each set K with all sets that precede it in $\mathcal{K}$. If J is a preceding set, K is eliminated if $J \subset K$ and J is eliminated if $J \supseteq K$. Some effort can be saved by ordering the sets of $\mathcal{K}$ according to increasing size; then K is not strictly contained in any preceding set. In any case, the number of set comparisons required to find $m[\mathcal{K}]$, if $\mathcal{K}$ consists of n sets, seems to be bounded above by some constant times n^2.

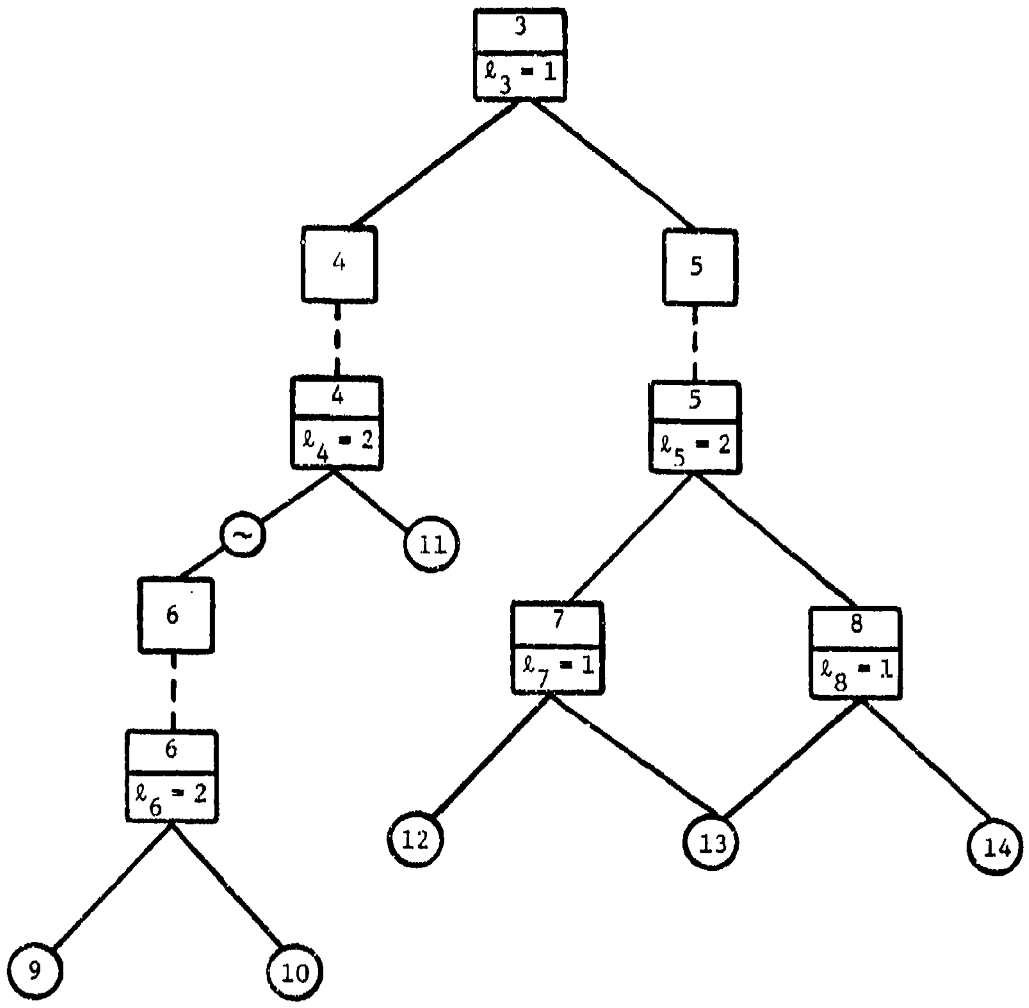
In practice, MOCUS and MICSUP algorithms often perform a great deal of minimization that could be avoided by isolating certain branches of the fault tree that have no basic nodes in common. Such is the general idea underlying this section.

I.3.1 Simple Module and Modular Subtree Definitions

We begin with some additional definitions regarding fault trees. If node w is a subnode of u , a *chain* from w to u is the set of nodes along an upward path from w to u . For instance, in the tree of Figure 1, the sets {12,7,5,3,2,1} and {12,7,5,1} are chains from node 12 to node 1. The number of nodes in the largest chain from w to u is denoted by $c_u(w)$, and $c_u(\cdot)$ is an integer valued function having u and the subnodes of u as its domain ($c_u(u) \equiv 1$ for any node u). In the example tree, $c_1(1) = 1$, $c_1(2) = 2$, $c_1(3) = 3$, $c_1(4) = 4$, $c_1(5) = 4$, $c_1(6) = 5$, etc.

Next, if v is a subnode of u , v is said to be a *simple module* for u if every chain from a basic subnode of v to u includes v . Node 5 is a simple module for node 1 in the example tree, since the basic subnodes of 5 are 12, 13, and 14, and the chains from these nodes to node 1 are [12,7,5,3,2,1] , [12,7,5,1] , [13,7,5,3,2,1] , [13,7,5,1] , [13,8,5,3,2,1] , [13,8,5,1] , [14,8,5,3,2,1] , and [14,8,5,1] , all of which include 5. It is helpful to think of the Boolean variables associated with a simple module as indicating the status of an independent subsystem; that is, given the status of the subsystem, the status of any component in the subsystem is irrelevant to the problem of determining whether the system itself is working. For instance, the values of x_{12} , x_{13} , and x_{14} are not important in determining the value of x_1 if the value of x_5 is known. Note that with this definition, node 4 is a simple module of node 3, but not of node 1, so the node or set of nodes must be specified for which a particular node is a simple module. Also, for any gate node u , all basic subnodes are trivially simple modules for u .

If node v is a simple module for node u and v is not a subnode of some other simple module for u , then v is a *largest* simple module for u . In Figure 1, the largest simple modules for node 1 are 5, 6, and 11, whereas those for node 3 are 4 and 5. It is easy to see that the largest simple modules for a node have no subnodes in common, a fact which motivates these definitions. The *modular subtree* for a gate node u consists of all nodes, along with the arcs joining these nodes, that appear in chains from the largest simple modules for u to u itself. Node u is thus the top node of its modular subtree. Figure 2 illustrates the modular subtrees for nodes 3, 4, 5, and 6 of the example tree. This definition of a modular subtree is related to the idea of an *independent branch* of the fault tree, introduced by Chatterjee [4].



3
$\ell_3 = 1$
4
5
4
$\ell_4 = 2$
5
$\ell_5 = 2$
~
11
6
7
$\ell_7 = 1$
8
$\ell_8 = 1$
6
$\ell_6 = 2$
12
13
14
9
10

FIGURE 2

I.3.2 Application of Simple Modules to Implicant Families

The above concepts may be readily associated with implicant families. For convenience, call event j a simple module for event i if node $|j|$ is a simple module for node $|i|$ in the fault tree, and the modular subtree for event i is the one having node $|i|$ at the top. Were the MOCUS or MICSUP method applied to the modular subtree for event i, with largest simple modules for i treated as basic events, the result would be a minimal implicant family M_i in terms of these largest simple modules. More suitable algorithms to find implicant families associated with modular subtrees are discussed in Section I.4, but for purposes here, nothing is lost by assuming that procedures similar to MOCUS or MICSUP are available to find families M_i.

For a given set Q of gate events, a *modular structure* for Q is a collection $\{M_j\}_{j \in M(Q)}$ of minimal implicant families in terms of largest simple modules for their respective index events, where the index set $M(Q)$ is the smallest set satisfying (1) $Q \subseteq M(Q)$; and (2) if j is a gate event in an implicant of some family M_i for $i \in M(Q)$, then $j \in M(Q)$. The modular structure for Q is just the smallest group of implicant families in terms of largest simple modules necessary to find a basic event implicant family for each event in Q. For the tree of Figure 1, $M(\{1\}) = \{1,5,6,-6\}$ and

$$M_1 = [\{5\},\{6\},\{-6,11\}]$$

$$M_5 = [\{13\},\{12,14\}]$$

$$M_6 = [\{9,10\}]$$

$$M_{-6} = [\{-9\},\{-10\}] \ .$$

However, $M(\{1,3\}) = \{1,3,4,5,6,-6\}$, so the modular structure for $\{1,3\}$ includes, in addition to the above families,

$$M_3 = [\{3\},\{4\}]$$

$$M_4 = [\{-6,11\}] \ .$$

Reliability evaluations for a fault tree are usually introduced by associating independent 0-1 random variables X_u $(\equiv 1 - X_{-u})$ with all basic nodes u . For a gate event i , with a minimal basic event implicant family $\mathcal{I}_i$, the random variable X_i is taken to be 1 if at least one set $I \in \mathcal{I}_i$ has $X_j = 1$ for all $j \in I$; otherwise, X_i is 0 . Under the assumption of probabilistic independence of basic node variables, variables for nodes that are largest simple modules of any particular gate event are also independent, since they have no common basic subnodes. Hence reliability evaluations for events $j \in M(\{i\})$ may be done by considering these events in upward order, and treating each family M_j as if it consisted of basic events. The number of minimal basic event implicants for i usually far exceeds the total number of implicants in all modular structure families $\{M_j\}_{j \in M(\{i\})}$.

Should basic event families be preferred for events in Q, they are easily obtained by selecting the events $j \in M(Q)$ in upward order and constructing I_j in the usual manner:

$$I_j \leftarrow \bigcup_{M \in M_j} \mathop{\mathsf{X}}_{m \in M} I_m$$

where, of course, $I_m \equiv [\{m\}]$ for m a basic event. Because the largest simple module events j have no subevents in common, minimization is unnecessary. In terrs of Boolean expressions, the modular structure for event 1 of the example tree yields,

$$x_{-6} = x_{-9} + x_{-10}$$

$$x_6 = x_9 x_{10}$$

$$x_5 = x_{13} + x_{12} x_{14}$$

$$\begin{aligned} x_1 &= x_5 + x_6 + x_{-6} x_{11} \\ &= (x_{13} + x_{12} x_{14}) + (x_9 x_{10}) + (x_{-9} + x_{-10}) x_{11} \\ &= x_{13} + x_{12} x_{14} + x_9 x_{10} + x_{-9} x_{11} + x_{-10} x_{11} . \end{aligned}$$

Size restriction may be employed when a MICSUP-type method is applied to modular subtrees for events in $M(Q)$. These subtrees should be devoid of complementing arcs. The family M_j' in the resulting collection $\{M_j'\}_{j \in M(Q)}$ will consist of all implicants of M_j that do not exceed the size limitation, and further elimination is done as basic event families I_j' are produced.

Of course, elimination based on an importance criterion for B is also feasible when basic event families are constructed. For $j \in M(Q)$, a subfamily I'_j is then constructed which consists of all important sets of the complete family I_j. However, it is useful to have an importance criterion for the set of all events that appear in at least one implicant of M_j. (Denote this set by $E(M_j)$.) Elimination can then be done when the MICSUP method is applied to the modular subtree for j. An importance criterion for $E(M_j)$ is easily obtained from a criterion for B by declaring a set $M \subseteq E(M_j)$ important iff the basic event family $\underset{m \in M}{X} I_m$ contains at least one important set. This is valid because the event sets $E(I_m)$, $m \in M$ are disjoint.

Such an importance criterion for $E(M_j)$ is actually quite easy to implement when real nonnegative values $\iota(k)$ are available for all basic events and an importance function f is given in terms of these values. Suppose, for instance, that $0 \leq \iota(k) \leq 1$ for $k \in B$, and for I a set of basic events, let $f(I) = \prod_{k \in I} \iota(k)$. For $j \in M(Q)$ we construct the subfamilies M'_j in upward order, that is, M'_j is, constructed following all families M'_k for k a subevent of j. When the family M'_j is constructed, for each gate event $m \in E(M_j)$ which is a largest simple module for j, a value $\iota(m)$ will be available from a previous computation unless $M'_m = \emptyset$. If the MICSUP method is applied to the modular subtree to find M'_j, then all sets generated will be in terms of largest simple modules for j, and only important sets need be retained, where in this case a set M is important iff $M'_m \neq 0$ for each $m \in M$ and (2) (if condition (1) holds)

$$\prod_{m \in M} \iota(m) > c$$

for a fixed critical value c. When the family M_j' has been found, if $M_j' \neq \emptyset$, $\iota(j)$ is determined by the computation

$$\max_{M \in M_j'} \left(\prod_{m \in M} \iota(m) \right) .$$

For $M_j' = \emptyset$, $\iota(j)$ is left undefined. It is quite easy to show that a set $M \subseteq E(M_j)$ satisfies (1) and (2) iff there is a basis event set $I \in \underset{m \in M}{X} I_m$ satisfying

$$\prod_{k \in I} \iota(k) > c .$$

This scheme can also be employed to yield a more efficient technique for size elimination than simply restricting implicants in modular structure families to a fixed maximum size. Let $\sigma(k) \equiv 1$ for each basic event k and let f be the importance function defined for a set I of basic events as $f(I) = \sum_{k \in I} \sigma(k)$. As in the procedure above, when M_j' is constructed, relevant values $\sigma(m)$ for largest simple module gate events $m \in E(M_j)$ will be available from earlier computations. A set $M \subseteq E(M_j)$ is now considered important iff (1) $M_m' \neq \emptyset$ for each $m \in M$, and (2) (if condition (1) holds)

$$\sum_{m \in M} \sigma(m) \leq c$$

for a fixed integer value c. From the family M'_j, if $M'_j \neq \emptyset$, $\sigma(j)$ computed as

$$\min_{M \in M'_j} \left(\sum_{m \in M} \sigma(m) \right) .$$

For any set $M \in M'_j$ in this case, there is at least one basic event set $I \in \mathop{X}_{m \in M} \mathcal{I}_m$ having no more than c elements. We call the particular criterion discussed in this paragraph *modular size importance*.

I.3.3 A Method for Identifying Modular Subtrees

For an arbitrary gate node u, let $G_u \cup L_u$ be the set of nodes in the modular subtree for u, where L_u consists of all nodes that are largest simple modules for u and $L_u \cap G_u = \emptyset$; G_u is never empty, since $u \in G_u$.

Finding sets G_u and L_u is not difficult computationally. The technique described here makes use of particular sets of *replicated* nodes, nodes which have more than a single leaving arc; for instance 5, 6 and 13 are the replicated nodes of the Figure 1 tree. For any node v, let R_v consist of all replicated subnodes of v, as well as v itself if v is replicated. The set of replicated subnodes of v is just

$$\bigcup_{w \in S_v} R_w ,$$

where S_v is the set of immediate subnodes of v.

The following procedure determines the set L_u of largest simple modules and the set G_u for an arbitrary gate node u :

<u>LSM</u>

```
0.  $z \leftarrow 0$ , $T \leftarrow \{u\}$ , $L_u \leftarrow \emptyset$ , $G_u \leftarrow \emptyset$ .
1.  If $T = \emptyset$ , stop.  Otherwise $z \leftarrow z + 1$ ,
    $T \leftarrow \{v \mid v \in T , c_u(v) \neq z\} \cup \{S_v \mid v \in T , c_u(v) = z\}$ ,
    $G_u \leftarrow G_u \cup \{v \mid v \in T , c_u(v) = z\}$ .
2.  $L \leftarrow \left\{ v \mid v \in T , c_u(v) = z + 1 , R_v \cap \left( \bigcup_{\substack{w \in T \\ w \neq v}} R_w \right) = \emptyset \right\}$
    $L_u \leftarrow L_u \cup L$ , $T \leftarrow T - L$ .  Go to 1.
```

As an illustration, we find the largest simple modules of the top node of the tree of Figure 1:

$$R_{14} , R_{12} , R_{11} , R_{10} , R_9 = \emptyset$$

$$R_{13} , R_7 , R_8 = \{13\}$$

$$R_6 = \{6\}$$

$$R_5 = \{5,13\}$$

$$R_4 = \{4,6\}$$

$$R_3 , R_2 , R_1 = \{4,5,6,13\}$$

$z \leftarrow 0$ $\quad T \leftarrow \{1\}$ $\quad L_1 \leftarrow \emptyset$ $\quad G_1 \leftarrow \emptyset$

$z \leftarrow 1$ $\quad T \leftarrow \emptyset \cup \{2,5,6\}$

$G_1 \leftarrow \emptyset \cup \{1\}$

$L \leftarrow \emptyset$

$z \leftarrow 2$ $\quad T \leftarrow \{5,6\} \cup \{3,4\}$

$G_1 \leftarrow \{1\} \cup \{2\}$

$L \leftarrow \emptyset$

$z \leftarrow 3$ $\quad T \leftarrow \{4,5,6\} \cup \{4,5\}$

$G_1 \leftarrow \{1,2\} \cup \{3\}$

$L \leftarrow \{5\}$ (since $R_5 \cap (R_4 \cup R_6) = \emptyset$)

$L_1 \leftarrow \emptyset \cup \{5\}$

$T \leftarrow \{4,5,6\} - \{5\}$

$z \leftarrow 4$ $\quad T \leftarrow \{6\} \cup \{6,11\}$

$G_1 = \{1,2,3\} \cup \{4\}$

$L \leftarrow \{6,11\}$ (since $R_{11} = \emptyset$)

$L_1 = \{5\} \cup \{6,11\}$

$T \leftarrow \{6,11\} - \{6,11\}$

Stop.

Essentially, the method proceeds down the subtree with top node u, and the set T and L involve nodes successively further from u with increasing z. An examination of Step 2 shows that if v is a subnode of u, unless v is a subnode of a largest simple module for u, then v will eventually appear in the table

T formed in Step 2 for some value of z, say z'. If $z' \neq c_u(v) - 1$ then v is retained in each T for successive values of $z = z', z' + 1, \ldots, c_u(v) - 1$. When $z = c_u(v) - 1$ in Step 3, then v is tested to see if it is a largest simple module for u. If it is, v is included in L_u and removed from T; otherwise, S_v replaces v in the next T formed in Step 2 (for $z = c_u(v)$) and v is included in G_u.

The validity of this procedure is based on the following easily established facts:

1. Given any two fault tree gate nodes v and w, neither being a subnode of the other, then $R_v \cap R_w = \emptyset$ if and only if v and w have no basic subnodes in common.
2. For any z, $T \cup L_u$ contains at least one node of every chain from a basic subnode of u to u itself; moreover, for each $v \in T$ and $w \in L_u$, $R_v \cap R_w = \emptyset$.
3. For $v \in T$, $z = c_u(v) - 1$, there is no $w \in T$ such that v is a subnode of w.
4. For $v \in T$, v a simple module for u, there is no $w \in T$ such that w is a subnode of v.

An effective method of constructing the set L in Step 3 deserves mention. Let $\underline{r} = (r_1, \ldots, r_n)$ be a vector having the same number of components as fault tree nodes. The set T is taken to be ordered in some arbitrary manner and for $v, w \in T$ we write $v \succ w$ if v precedes w in T. Node u is again the subtree top node.

```
Ø.  $L^1 \leftarrow \emptyset$ . For each $w \in R_u$ , $r_w \leftarrow 0$ in $\underline{r}$ .
1.  If all nodes of $T$ have been considered in this step,
    stop. Otherwise select the next element $v \in T$ in the
    order $\succ$ .
2.  If $z \neq c_u(v) - 1$ , go to 3. Otherwise, if $r_w = 0$ for
    all $w \in R_v$ , then $L^1 \leftarrow L^1 \cup \{v\}$ .
3.  $r_w \leftarrow v$ for all $w \in R_v$ . Go to 2.
```

This procedure constructs the set

$$L^1 = \{v \mid v \in T \text{ , } c_u(v) = z + 1 \text{ , and } R_w \cap R_v = \emptyset \text{ for all } w \succ v \text{ in } T\} .$$

The same procedure may be applied with the modification that the elements of T be considered in reverse order to obtain

$$L^2 = \{v \mid v \in T \text{ , } c_u(v) = z + 1 \text{ , and } R_w \cap R_v = \emptyset \text{ for all } w \prec v \text{ in } T\} .$$

Then $L = L^1 \cap L^2$.

We sometimes require that the largest simple modules be known for each gate node of the fault tree, so the LSM procedure must be applied for every $u \in G$. However, calculating $c_u(w)$ for all subnodes w of u for each $u \in G$ is wasteful, and a more efficient method can be suggested. This method is motivated by two simple facts: First, if G_u and L_u are available for some gate node u , and we wish to find G_v and L_v for some $v \in G_u$, then it is only necessary to calculate $c_v(\cdot)$ for subnodes of v in $G_u \cup L_u$, since $G_v \cup L_v \subset G_u \cup L_u$. Secondly, if v is any gate node which is a simple module for u , then $c_v(w) = c_u(w) - c_u(v)$ for any subnode w of v . In the following statement of the method, it is assumed that a downward order has been determined for the set G of all gate nodes.

<u>MODS</u>

```
0. For all $w \in G$ , $N_w \leftarrow G \cup B$ .
1. If all nodes in $G$ have been considered in this step,
   stop. Otherwise, select the next node $u \in G$ in downward
   order.
2. If $L_u$ and $G_u$ have been found, go to 1. Otherwise
   calculate $c_u(\cdot)$ for all subnodes of $u$ in $N_u$ , and
   $M \leftarrow \{u\}$ .
   a. If $L_v$ and $G_v$ have been found for all nodes $v \in M$ ,
      go to 1. Otherwise select $v \in M$ for which $L_v$ and
      $G_v$ are not available.
   b. Find $L_v$ and $G_v$ using LSM procedure, noting that
      either $v \equiv u$ or $v$ is a simple module for $u$ , so
      $c_v(w) = c_u(w) - c_u(v)$ for all subnodes $w$ of $v$ .
   c. $M \leftarrow M \cup \{w \mid w \in L_v , w$ a gate node$\}$ . For each
      $w \in G_v - \{v\}$ for which $G_w$ is not available,
      $N_w \leftarrow G_v \cup L_v$ . Go to 2a.
```

Note that substeps 2a thru 2c are repeated until *all* simple modules have been found for node u chosen in Step 1.

The process of determining largest simple modules for each gate node of the tree of Figure 1 is illustrated:

$N_w \leftarrow \{1,2,3,4,5,6,7,8,9,10,11,12,13,14\}$, $w \in G$.

Calculate $c_1(\cdot)$ for subnodes of node 1 in N_1 . $M \leftarrow \{1\}$.

$L_1 = \{5,6,11\}$, $G_1 = \{1,2,3,4\}$

$M \leftarrow \{1\} \cup \{5,6\}$

$N_w \leftarrow \{1,2,3,4,5,6,11\}$, $w \in \{2,3,4\}$

$L_5 = \{12,13,14\}$, $G_5 = \{5,7,8\}$

$N_w \leftarrow \{5,7,8,12,13,14\}$, $w \in \{7,8\}$

$L_6 = \{9,10\}$, $G_6 = \{6\}$.

Calculate $c_2(\cdot)$ for subnodes of node 2 in N_2 . $M \leftarrow \{2\}$.

$L_2 = \{4,5\}$, $G_2 = \{2,3\}$

$M \leftarrow \{2\} \cup \{4,5\}$

$N_3 \leftarrow \{2,3,4,5\}$

$L_4 = \{6,11\}$, $G_4 = \{4\}$

$M \leftarrow \{2,4,5\} \cup \{6\}$

L_5 , G_5 found previously

L_6 , G_6 found previously.

Calculate $c_3(\cdot)$ for subnodes of node 3 in N_3 . $M \leftarrow \{3\}$.

$L_3 = \{4,5\}$, $G_3 = \{3\}$

$M \leftarrow \{3\} \cup \{4,5\}$

L_4 , G_4 found previously

L_5 , G_5 found previously.

L_4 , G_4 found previously.

L_5 , G_5 found previously.

L_6 , G_6 found previously.

Calculate $c_7(\cdot)$ for subnodes of node 7 in N_7 . $M \leftarrow \{7\}$.

$$L_7 = \{12,13\} \text{ , } G_7 = \{7\} \text{ .}$$

Calculate $c_8(\cdot)$ for subnodes of node 8 in N_8 . $M \leftarrow \{8\}$.

$$L_8 = \{13,14\} \text{ , } G_8 = \{8\} \text{ .}$$

Stop.

I.4 Obtaining Implicant Families Associated with Modular Subtrees

Subsections I.4.1, I.4.2, and I.4.3 each suggest a technique for deriving a minimal implicant family M_i associated with the modular subtree with top event i . If the subtree involves complementing arcs, then the complete families, say M_i , $\hat{M}_i$, and $\hat{\hat{M}}_i$, generated by each of these three methods may all be different, though it will be true that for every $\underline{x}$, $/M_i/\underline{x} = /\hat{M}_i/\underline{x} = /\hat{\hat{M}}_i/\underline{x}$. The families produced by method MSDOWN of I.4.1 and method MSUP of I.4.2 need not be prime implicant families for i when the subtree has complementing arcs. The Nelson method of I.4.3 always generates a prime implicant family or subfamily of all prime implicants that agree with an importance criterion or size restriction; however, this method will often be less efficient than MSDOWN or MSUP when applied to a large subtree.

Subsection I.4.4 speculates on the relative suitability of these algorithms for particular applications.

I.4.1 The MSDOWN Method

The spirit of this method (Modular Subtree Downward) is akin to that of MOCUS, but MSDOWN is more intricate and more efficient for most applications. The algorithm makes use of the concept presented in Section I.1 of the dual of a family of sets of positive and negative integers. For purposes here, this concept requires some additional comment, which is introduced by way of an example.

Consider the fault tree of Figure 3. Were the MOCUS algorithm applied to this tree, the process of constructing the minimal implicant family $\mathcal{I}_1$ would be represented by (with $\ell_e = 1$)

$$m\left[\underset{e\in\{2,3,4,5\}}{X} \mathcal{D}(\ell_e, D_e)\right],$$

where $D_2 = \{6,7,8,9\}$, $D_3 = \{7,8,-9,10\}$, $D_4 = \{7,8,11,12\}$, and $D_5 = \{9,10,13,14\}$. In a Boolean context, the state vector is $\underline{x} = (x_1, x_2, \ldots, x_{14})$, and the expression for $/\underset{e\in\{2,3,4,5\}}{X} \mathcal{D}(1,D_e)/\underline{x}$ is a product of sums,

$$(x_6 + x_7 + x_8 + x_9)(x_7 + x_8 + x_{-9} + x_{10})(x_7 + x_8 + x_{11} + x_{12})(x_9 + x_{10} + x_{13} + x_{14}).$$

So determining the product family and minimizing is essentially equivalent to expanding the above expression into a sum of products and eliminating nonminimal products, as well as products having complementary pairs of variables. Of the 256 products, 228 have no complementary pairs of variables, but only 16 products are minimal.

Though the tree of Figure 3 is contrived, and such trees do not often occur in practice, the point is that if an implicant H with

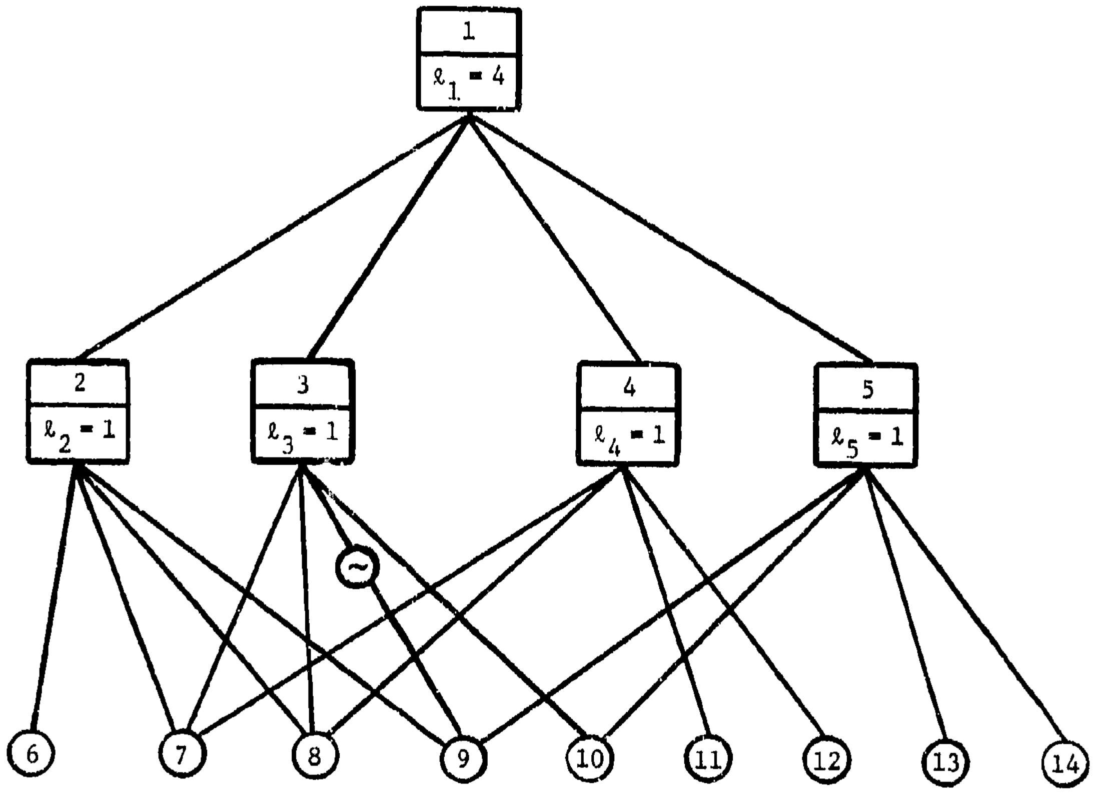


FIGURE 3

a moderate number of gate events appears at some time during application of the MOCUS procedure, the product family $\underset{e \in H}{X} \mathcal{D}(\ell_e, D_e)$ may be quite large, especially for a tree where a sizable proportion of the ℓ_e are 1 (OR relations). However, the family remaining after minimization may be relatively small if the immediate subevent sets D_e involve events associated with replicated nodes. This suggests that substantial effort could be avoided if the family

$$m \left[\underset{e \in H}{X} \mathcal{D}(\ell_e, D_e) \right]$$

could be found without generating all the nonminimal sets in the product family.

From the definition of the family $\mathcal{D}(\ell_e, D_e)$, it is clear from Section I.1 that $d[\mathcal{D}(\ell_e, D_e)] = \mathcal{D}(\#D_e - \ell_e + 1, D_e)$, where $\#D_e$ is the number of elements in D_e. Moreover, $d[d[\mathcal{D}(\ell_e, D_e)]] = \mathcal{D}(\ell_e, D_e)$. Thus by Proposition I.1.3,

$$d \left[\bigcup_{e \in H} \mathcal{D}(\#D_e - \ell_e + 1, D_e) \right] = m \left[\underset{e \in H}{X} \mathcal{D}(\ell_e, D_c) \right] .$$

The family in brackets on the left requires about the same amount of effort to construct as the families $\mathcal{D}(\ell_e, D_e)$ together. The algorithm given in Reference [17] finds the dual of an arbitrary family F and is well suited to our purposes here. Essentially, the algorithm generates the sets of $d[F]$ in groups, and minimization is required only among members of the same group; in fact, when finding the dual of $\bigcup_{e \in H} \mathcal{D}(\#D_e - \ell_e + 1, D_e)$ the algorithm bypasses minimization altogether if the sets D_e, $e \in H$ are pairwise disjoint.

In addition, the number of nonminimal sets appearing during construction of this dual will always be less than the number of such sets in $\underset{e\in H}{X} \mathcal{D}(\ell_e,S_e)$, usually many times less. Use of the dual algorithm in the manner suggested here to find I_1 for the tree of Figure 3 requires only 1/10 the computation time necessary to produce and minimize the product family. The difference in efficiency between the two methods becomes increasingly dramatic as the number of sets in the product family increases. It is not hard to devise examples where the dual algorithm generates the required minimal family quite easily, but formation of the product family is computationally impossible.

If $\underset{e\in H}{X} \mathcal{D}(\ell_e,D_e)$ is small, say fewer than 20 sets, the dual algorithm may require somewhat more computation time than forming and minimizing the product family, due to the comparatively large amount of computer code associated with the algorithm. However, in this case, the computation time required by either method is quite negligible; so in the MSDOWN method, it is not worth the trouble to bypass the dual algorithm and derive and minimize the product family whenever this family has fewer than 20 sets.

The steps below comprise the MSDOWN procedure applied to a modular subtree with top event i . The procedure requires that the set L_u of largest simple modules for $u = |i|$ be available, as well as the set G_u of subtree nodes which are not in L_u .

<u>MSDOWN</u>

```
Ø. $z \leftarrow 0$ , $\alpha \leftarrow 0$ , $u \leftarrow |i|$ , $\mathcal{H} \leftarrow [\{i\}]$ .
1. $z \leftarrow z + 1$ , $C^z \leftarrow \{v \mid v \in G_u , c_u(v) = z\}$ .
   $R^z \leftarrow \{v \mid v \in G_u \cup L_u , c_u(v) = z + 1 , v$ replicated$\}$ .
   If $C^z = \emptyset$ , $M_i \leftarrow \mathcal{H}$ and stop.
2. If all $H \in \mathcal{H}$ that intersect $C^z \cup (-C^z)$ have been
   considered previously in this step, go to 5.
   Otherwise, select $H \in \mathcal{H}$ with $H \cap (C^z \cup (-C^z)) \neq \emptyset$
   that has not been considered.
3. a. For each $e \in H$ if $e \in C^z \cup (-C^z)$ ,
         $J_e \leftarrow \mathcal{D}(\#D_e - \ell_e + 1, D_e)$ , and if $e \notin C^z \cup (-C^z)$ ,
         $J_e \leftarrow [\{e\}]$ .
   b. $\alpha \leftarrow \alpha + 1$ .
   c. $\mathcal{H}_\alpha \leftarrow d\left[\bigcup_{e \in H} J_e\right]$ .
   d. $\mathcal{H} \leftarrow [\mathcal{H} - [H]] \cup \mathcal{H}_\alpha$ , go to 2.
4. $\mathcal{R} \leftarrow [H \mid H \in \mathcal{H} , H \cap (R^z \cup (-R^z)) \neq \emptyset]$ .
   ($\mathcal{R}$ will thus contain all $H$ having a replicated
   subevent $e$ with $c_u(|e|) = z + 1$ .)
5. Partition sets of $\mathcal{R}$ into disjoint families $\{\mathcal{R}_\alpha\}_{\alpha \in A}$ ,
   where $\mathcal{R}_\alpha = \mathcal{R} \cap \mathcal{H}_\alpha$ and $A$ consists of all $\alpha$ such
   that $\mathcal{R}_\alpha \neq \emptyset$ .
6. $\mathcal{H} \leftarrow [\mathcal{H} - \mathcal{R}] \cup m\left[\bigcup_{\alpha \in A} \mathcal{R}_\alpha\right]$ , go to 1.
```

Following each execution of Step 1, events $e \in H$ for any implicant $H \in \mathcal{H}$ all satisfy $c_u(|e|) \geq z$. Steps 3a thru 3d select each implicant $H \in \mathcal{H}$ having at least one event e satisfying $c_u(|e|) = z$ and replace this implicant with a family $\mathcal{H}_\alpha$.

For $e \varepsilon H$, if we let $\mathcal{J}_e = \mathcal{D}(\#D_e - \ell_e + 1, D_e)$ for $c_u(|e|) = z$ and $\mathcal{J}_e = [\{e\}]$ otherwise, then

$$\mathcal{H}_\alpha = m\left[\underset{e\varepsilon H}{X} \mathcal{J}_e\right]$$

by our earlier remarks. Thus when Step 4 is begun, events $e \varepsilon H$ for any $H \varepsilon \mathcal{H}$ have $c_u(|e|) > z$. At this point, the population of events e with $c_u(|e|) = z + 1$ is greater than in all families $\mathcal{H}$ previously constructed, since no substitution for these events has yet occurred. The events which correspond to replicated nodes of the subtree and satisfy $c_u(|e|) = z + 1$ are likely to be found in nonminimal sets of $\mathcal{H}$. So we assign sets containing such replicated events to the family $\mathcal{R}$, and minimize only this portion of $\mathcal{H}$. Moreover, for any family $\mathcal{H}_\alpha$ constructed in Step 3, the intersection of $\mathcal{H}_\alpha$ and $\mathcal{R}$ is minimal (though it may be empty), because $\mathcal{H}_\alpha$ is minimal. Each family of the partition in Step 5 is thus minimal, so the indicated minimization only requires comparison of each set of a family $\mathcal{R}_\alpha$ with all sets in preceding families.

It is intuitive but not obvious that the minimization scheme in this algorithm insures $\mathcal{M}_1$ will be minimal. This is the case, but to establish this fact rigorously is tedious and not particularly instructive, so we do not consider the proof.

Figure 4 shows the modular subtree for the top node of the Figure 1 tree. The following example derives the minimal implicant family $\mathcal{M}_1$ for this subtree using Boolean variables. The integer in parentheses following a term is the value α associating the corresponding implicant with the family $\mathcal{H}_\alpha$. The families $\mathcal{R}$ of Step 4 and those of the partitions of Step 5 are also indicated.

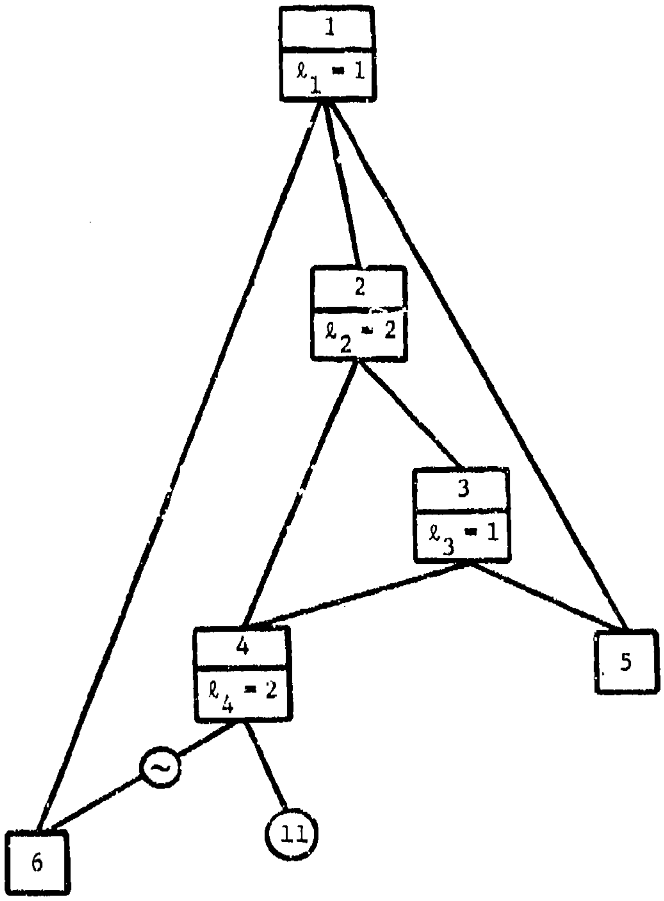


FIGURE 4

$z \leftarrow 1 \qquad C^1 \leftarrow \{1\} \qquad R^1 \leftarrow \emptyset$

$$\left.\begin{array}{l} x_1 \\ + x_2(1) \\ + x_5(1) \\ + x_6(1) \end{array}\right\} \leftarrow (/d[[\{1,2,3\}]]/\underline{x})$$

$\mathcal{R} = \emptyset$

$z \leftarrow 2 \qquad C^2 \leftarrow \{2\} \qquad R^2 \leftarrow \emptyset$

x_2

$+ x_5$

$+ x_6(1)$

$+ x_3x_4(2) \leftarrow (/d[[\{3\},\{4\}]]/\underline{x})$

$\mathcal{R} = \emptyset$

$z \leftarrow 3 \qquad C^3 \leftarrow \{3\} \qquad R^3 \leftarrow \{4,5\}$

$x_5(1)$

$+ x_6(1)$

$+ x_3x_4$

$+ x_4(3) \leftarrow (/d[[\{4\},\{4,5\}]]/\underline{x})$

$\mathcal{R} = [\{4\},\{5\}]\ ,\ \mathcal{R}_1 = [\{5\}]\ ,\ \mathcal{R}_3 = [\{4\}]$

$z \leftarrow 4 \qquad C^4 \leftarrow \{4\} \qquad R^4 \leftarrow \{6\}$

$x_5(1)$

$+ x_6(1)$

$+ x_4$

$+ x_{-6}x_{11}(4) \leftarrow (/d[[\{-6\},\{11\}]]/\underline{x})$

$\mathcal{R} = [\{6\},\{-6,11\}]\ ,\ \mathcal{R}_1 = [\{6\}]\ ,\ \mathcal{R}_4 = [\{-6,11\}]$

$z \leftarrow 5 \qquad C \leftarrow \emptyset \qquad$ Stop.

The expression associated with $/M_1/\underline{x}$ is thus $x_5 + x_6 + x_{-6}x_{11}$.

I.4.2 The MSUP Method

The MSUP algorithm resembles MICSUP confined to a modular subtree. MSUP is particularly suited to applications where only a subfamily of important implicants or those not exceeding a fixed size is required for the subtree with top event i , rather than a complete family M_i .

As with MSDOWN, the MSUP method utilizes the set L_u of largest simple modules for the subtree top node $u = |i|$, as well as the set G_u of subtree nodes not in L_u . In addition, MSUP requires that sets L_v be available for all $v \in G_u$; thus, prior to deriving the modular structure families $\{M_j\}_{j \in M(Q)}$ using MSUP, it is convenient to apply the MODS algorithm of Subsection I.3.3 to determine the largest simple modules of every fault tree gate event. Finally, MSUP calls on a "downward" type subalgorithm designated as ORDOWN (substitution for OR-relations, DOWNward). Since ORDOWN has much in common with the MSDOWN method of the previous subsection, we first discuss this subalgorithm.

ORDOWN, like MSDOWN, obtains an implicant family N_j for j a top event of a modular subtree. However, the events in implicants of the family N_j need not correspond to largest simple modules for the subtree top node $v = |j|$. The method is outlined as follows:

ORDOWN

```
0.  $\alpha \leftarrow 0$ , $v \leftarrow |j|$ , $H \leftarrow [\{j\}]$ .
1.  $C \leftarrow \{v\} \cup \{w \mid w \in G_v , \ell_w = 1\}$ .
2.  If all $H \in H$ that intersect $C \cup (-C)$ have been considered
    previously in this step, go to 4. Otherwise select $H \in H$
    with $H \cap (C \cup (-C) \neq \emptyset$ that has not been considered.
3.  a.  For each  $e \in H$ , if  $e \in C \cup (-C)$
            $\mathcal{J}_e \leftarrow \mathcal{D}(\#D_e - \ell_e + 1, D_e)$ , and if  $e \notin C \cup (-C)$ ,
            $\mathcal{J}_e \leftarrow [\{e\}]$ .
    b.  $\alpha \leftarrow \alpha + 1$ .
    c.  $\mathcal{H}_\alpha \leftarrow d\left[\bigcup_{e \in H} \mathcal{J}_e\right]$ .
    d.  $\mathcal{H} \leftarrow [\mathcal{H} - [H]] \cup \mathcal{H}_\alpha$  and go to 2.
4.  Partition sets of $\mathcal{H}$ into disjoint families $\hat{\mathcal{H}}_\alpha = \mathcal{H} \cap \mathcal{H}_\alpha$ and
    let A consist of all $\alpha$ such that $\hat{\mathcal{H}}_\alpha \neq \emptyset$ .
5.  $\mathcal{N}_j \leftarrow m\left[\bigcup_{\alpha \in A} \hat{\mathcal{H}}_\alpha\right]$ and stop.
```

Note that each $\hat{\mathcal{H}}_\alpha$ is minimal, since $\mathcal{H}_\alpha$ arises from a single application of the dual algorithm; thus, the minimization in Step 5 involves comparing sets in $\hat{\mathcal{H}}_\alpha$ only with sets in preceding families of the union.

The form of this method is somewhere between MOCUS and MSDOWN, but its important feature is the set C of Step 1 which controls event substitution in implicants of $\mathcal{H}$. Substitution for the top event j is always done, but a subsequent event e appearing in sets of $\mathcal{H}$ that is not a largest simple module for j may only be replaced by $\mathcal{D}(\ell_e, D_e)$ if $\ell_e = 1$, that is, if x_e is represented by an OR relation between immediate subevent variables. One effect of this restriction is that no set of the family $\mathcal{N}_j$ will contain more events than a set in the top event definition family, $\mathcal{D}(\ell_j, D_j)$, though $\mathcal{N}_j$ will usually contain more sets than the definition family. A second effect is that events in implicants of $\mathcal{N}_j$ are more likely to correspond to largest simple modules for j than events in D_j , though it may happen, of course, that N_j and $\mathcal{D}(\ell_j,D_j)$ are the same. The motivation for producing families N_j for selected gate events j of the modular subtree will be discussed in connection with the MSUP method.

For a large modular subtree with top event j, sets of N_j will usually involve only events for nodes near the top of the subtree, and in such a case, it is to be expected that N_j will contain many fewer sets that the family M_j produced by the MSDOWN method. For this reason, the more elaborate minimization scheme of MSDOWN has not been included in ORDOWN. However, MSDOWN can be modified to produce the family N_j instead of M_j by changing the formation of the set C^z in Step 1 of that algorithm.

For event 1 of the Figure 1 tree, ORDOWN proceeds in this fashion:

$$C \leftarrow \{1,3\}$$

$$\begin{array}{l} x_1 \\ \left.\begin{array}{l} + x_2(1) \\ + x_5(1) \\ + x_6(1) \end{array}\right\} \leftarrow (/d[[\{1,2,3\}]]/\underline{x}) \end{array}$$

(No set in $H = [\{2\},\{5\},\{6\}]$ intersects C).

$$H_1 = \hat{H}_1 = [\{2\},\{5\},\{6\}] \; .$$

Stop.

The expression associated with $/N_1/\underline{x}$ is $x_2 + x_5 + x_6$. Hence, for this example $N_1 = \mathcal{D}(\ell_1, D_1)$. But for event 5, ORDOWN gives $/N_5/\underline{x}$ as $x_{12}x_{14} + x_{13}$.

To find the minimal family M_i in terms of largest simple module for event i , the steps of the MSUP algorithm are as follows:

<u>MSUP</u>

```
0.  $F \leftarrow \{i\}$ , $u \leftarrow |i|$ .
1.  If all events $j \in F$ have been considered previously
    in this step, go to 4.  Otherwise select $j \in F$ not yet
    considered.
2.  Determine the family $N_j$ by applying algorithm ORDOWN
    to the modular subtree with top event $j$ .
3.  $F \leftarrow F \cup \{e \mid e \in E(N_j)$ , $e$ not a largest simple module for $i\}$ .
    Go to 1.
4.  Consider events $j \in F$ in upward order (so any event of
    F follows its subevents), constructing families $K_j$ in
    this manner:  If all events in $E(N_j)$ are largest
    simple modules for $j$ ,

        $K_j \leftarrow \bigcup_{N \in N_j} \mathop{X}_{n \in N} K_n$ ,

    where $K_n \equiv [\{n\}]$ if $n \in L_u$ .  If not all events in
    $E(N_j)$ are largest simple modules for $j$ ,

        $K_j \leftarrow m\left[\bigcup_{N \in N_j} \mathop{X}_{n \in N} K_n\right]$ .

5.  $M_i \leftarrow K_i$ and stop.
```

$E(N_j)$ appearing in Steps 3 and 4 is the set of all events appearing in at least one implicant of N_j . Also, though the facility for implicant elimination based on an importance criterion or size limitation has not been included in this outline of MSUP, elimination may be carried out in Step 4 just as indicated in Subsections I.2.2 and I.3.2 with regard to the MICSUP method.

The families K_j generated in Step 4 are all in terms of largest simple modules for event i . In fact, if in Step 2 the ORDOWN procedure is ignored and $N_j \leftarrow \mathcal{D}(\ell_j, D_j)$, then the resulting method is the MICSUP procedure applied to the modular subtree for event i , with the exception that information concerning simple modules guides minimization in Step 4. The incentive for obtaining N_j from the ORDOWN algorithm is threefold: First, sets of N_j are more likely to contain only simple modules for j than sets of $\mathcal{D}(\ell_j, D_j)$, so there is less likelihood that minimization will be required when K_j is constructed. Secondly, since we are ultimately interested in the implicant family K_i ($= M_i$) , construction of implicant families for other events in the subtree for event i should be avoided if possible. Use of ORDOWN usually leads to a smaller set of events F at the beginning of Step 4 than if N_j were set to $\mathcal{D}(\ell_j, D_j)$, since an OR gate event $e \in D_j$, not a simple module for some other event in F , would not appear in F . Finally, the sets of N_j are no larger than those of $\mathcal{D}(\ell_j, D_j)$, so implicant elimination based on size or importance in Step 4 is no more difficult than in MICSUP.

For event 1 of the tree of Figure 1, repeating Steps 1, 2, and 3 of MSUP yields families N_1, N_2, and N_4 represented by the Boolean expressions:

$$x_1 = x_2 + x_5 + x_6$$

$$x_2 = x_4$$

$$x_4 = x_{-6}x_{11} .$$

The set F (in proper order) is $\{4,2,1\}$. Since 4 is a simple module for 2, minimization is only done when M_1 is found:

$$x_4 = x_{-6}x_{11}$$

$$x_2 = x_4$$

$$= x_{-6}x_{11}$$

$$x_1 = x_{-6}x_{11} + x_5x_6 .$$

I.4.3 The Nelson Method

Associated with any given fault tree is a *dual tree*, which differs from the original, or *primal tree*, only in the value of gate node logic indicators. If ℓ_u is the logic indicator for node u of the primal tree, then $\#D_u - \ell_u + 1$ is the corresponding logic indicator for the same node of the dual tree. Of course, for trees having only AND and OR logic, the dual tree is easily obtained from the primal by changing each AND gate to an OR gate and vice-versa. Since the defining families $\mathcal{D}(\ell_u, D_u)$ and $\mathcal{D}(\#D_u - \ell_u + 1, D_u)$, for gate event u of the primal and dual trees, are dual families, Proposition I.1.1 of Section I.1 indicates that for all $\underline{x}$,

$$/\mathcal{D}(\#D_u - \ell_u + 1, D_u)/(\underline{1} - \underline{x}) = 1 - /\mathcal{D}(\ell_u, D_u)/\underline{x} .$$

This holds for all $u \varepsilon G$, so for any vector $\underline{x}$ consistent with the primal tree, in the sense of Section I.2, the vector $\underline{1} - \underline{x}$ is consistent with the dual tree; that is, for all $u \varepsilon G$,

$$/\mathcal{D}(\#D_u - \ell_u + 1, D_u)/(\underline{1} - \underline{x}) = 1 - x_u .$$

Were the MSDOWN (or MSUP) method applied to the modular subtree for event i in each of these trees to obtain a family M_i^d for the dual tree and a family M_i for the primal, then for all $\underline{x}$ consistent with the primal tree it would be the case that

$$/M_i^d/(\underline{1} - \underline{x}) = 1 - /M_i/\underline{x} .$$

So again by Proposition I.1.1, the dual family, $d\left[M_i^d\right]$, associated with M_i^d would satisfy, for all $\underline{x}$ consistent with the primal tree,

$$/d\left[M_i^d\right]/\underline{x} = /M_i/\underline{x} .$$

Thus we see another way to construct a minimal implicant family M_i from the modular subtree for event i : Apply the MSDOWN (or MSUP) algorithm to obtain a complete minimal family M_i^d for the dual modular subtree for event i, and then construct the dual family, $d\left[M_i^d\right]$, associated with M_i^d.

This procedure may not always be successful in practice. The first problem involves obtaining M_i^d ; this may not be possible if the modular subtree for i is large, since M_i^d must be a complete minimal family, and a subfamily of important or size restricted sets is not adequate. Secondly, even when M_i^d can be generated, construction of $d\left[M_i^d\right]$ may be difficult. There is a well-known argument that a "good" algorithm for finding the dual family for an arbitrary family will probably never be devised [1], [15]; a "good" algorithm would be such that the effort required could be bounded in all cases by a fixed polynomial in the number of sets in the dual family or the number of elements composing these sets. This, however, is not intended to suggest that all algorithms for constructing dual families are equally "bad."

The dual algorithm given in [17] and previously recommended for use in MSDOWN and ORDOWN methods has worked well for obtaining $d\left[M_i^d\right]$ in a number of applications, some involving quite large modular subtrees. This algorithm also permits set elimination based on importance and size criteria to be utilized to considerable advantage in constructing a subfamily of all important sets of $d\left[M_i^d\right]$ or those not exceeding a fixed size. In fact, if the complete family M_i^d is available, adequate size and importance restrictions can almost always be chosen to insure that some subfamily of $d\left[M_i^d\right]$ will be found with a moderate amount of computational effort.

In some instances where this method has been applied to large subtrees, the process of obtaining M_i^d and then implicants of $d\left[M_i^d\right]$ not exceeding fixed size has proven to be several times faster than employing the MSUP algorithm to find the family M_i' with the same size restriction. These example subtrees did not contain complementing arcs; thus, in each case the subfamilies generated by the two methods were the same. One subfamily involved 1000 sets, so the difference in effort required by the two methods can be significant. However, it is well to note that families M_i^d for the dual subtrees all had less than 50 sets, though some of these sets consisted of more than 25 events.

When the modular subtree for an event i contains complementing arcs, $d\left[M_i^d\right]$ will usually not be the same family as that produced by the MSDOWN or MSUP method. For instance, $M_1^d = [\{5,6,11\}]$ for event 1 of the example tree of Figure 1, so $d\left[M_1^d\right] = [\{5\},\{6\},\{11\}]$, which differs from $M_1 = [\{5\},\{6\},\{-6,11\}]$ obtained by MSDOWN and MSUP. The family $d\left[M_1^d\right]$ is a prime implicant family for the Boolean function $/[\{5\},\{6\},\{-6,11\}]/$ so $d\left[M_1^d\right]$ is a prime implicant family for event 1 in the sense of Subsection I.2.1. The method suggested here may be recognized as an adaptation of "Nelson's Algorithm" [11] for finding a prime implicant family for the Boolean function $/F/$, given an arbitrary family F of subsets of $U \cup (-U)$ (where U , as usual, is some set of consecutive positive integers, say $\{1, \ldots, q\}$). It turns out that $P = d[d[F]]$ is the required family. One way to prove this is to show that $d[F]$ is a prime implicant family for the function $/d[F]/$, which can be done by demonstrating that if there is a proper subset of some $P \varepsilon d[F]$ such that this subset implies $/d[F]/$, then there is an $\underline{x} = (x_1, \ldots, x_q)$ such that both $/d[F]/\underline{1} - \underline{x} = 1$ and $/F/\underline{x} = 1$, contradicting Proposition I.1.1. Our version of this technique is to find $d[d[M_i]]$ by replacing $d[M_i]$ with the minimal family M_i^d , obtained through application of MSDOWN or MSUP to the dual modular subtree for i . Though in general $M_i^d \neq d[M_i]$, it is true that $d\left[M_i^d\right] = d[d[M_i]]$.

Letting P_j represent the prime implicant for event j in terms of largest simple modules for j , the collection $\{P_j\}_{j \varepsilon M(Q)}$ may now be derived by utilizing the Nelson method when the modular subtree for j involves complementing arcs, and any one of the methods MSDOWN, MSUP, or Nelson when complementing arcs are absent. Suppose families in terms of basic events are generated in the manner of Subsection I.3.2; that is, events $j \varepsilon M(Q)$ are considered in upward order and each basic event family I_j is generated by

$$I_j \leftarrow \bigcup_{P \varepsilon P_j} \mathop{\mathrm{X}}_{p \varepsilon P} I_p \ .$$

Intuitively, it would seem that these basic event families should also be prime implicant families, and this is in fact the case. Also, since the dual algorithm is capable of constructing a subfamily P'_j consisting of all sets of $d\left[M_j^d\right]$ satisfying a size or importance restriction, the remarks of Subsection I.3.2 extend in an obvious way to fault trees with complementing arcs. Thus a collection $\{P'_j\}_{j \in M(Q)}$ of subfamilies may be obtained such that P'_j is a basic event subfamily of all important prime implicants or those not exceeding some fixed size.

To conclude this subsection, we note that the efficiency of finding a prime implicant family in the manner suggested by $d[d[F]]$ greatly depends on the particular technique utilized to construct dual families. Sometimes the name "Nelson's Algorithm" is applied to a detailed procedure, also called the method of *double complements*, which is not noted for being very efficient. This procedure begins with a Boolean expression in sum-of-products form, for example,

$$x_1x_2 + \bar{x}_2x_3x_4 + x_3\bar{x}_4 \; .$$

The expression is complemented and, using DeMorgan's Law, converted to a product of sums:

$$(\bar{x}_1 + \bar{x}_2)(x_2 + \bar{x}_3 + \bar{x}_4)(\bar{x}_3 + x_4) \; .$$

Next, a sum of products is obtained by expanding and eliminating products which are not minimal or contain complementary pairs of variables:

$$\bar{x}_1x_2x_4 + \bar{x}_1\bar{x}_3 + \bar{x}_2\bar{x}_3 \; .$$

This last expression is again complemented, repeating the above steps to yield

$$x_1x_2 + x_1x_3 + \bar{x}_2x_3 + \bar{x}_4x_3 \ .$$

In our notation, $P = [\{1,2\},\{1,3\},\{-2,3\},\{3,-4\}]$ is then a prime implicant family for the function $/F/$, where $F = [\{1,2\},\{-2,3,4\},\{3,-4\}]$.

The effort required by the double complement method increases very rapidly as the size of F increases. Hulme and Worrell [8] considered the following sum of twenty products:

$$\begin{aligned} & x_1x_6x_7x_9 + \bar{x}_2x_6x_7x_8 + x_1\bar{x}_3x_4x_6 + x_1x_6\bar{x}_7x_8 \\ & + \bar{x}_1\bar{x}_4\bar{x}_5x_7 + x_3x_4x_7x_8 + x_3\bar{x}_6\bar{x}_8x_9 + x_1x_2\bar{x}_3x_9 \\ & + \bar{x}_1\bar{x}_3x_6x_9 + x_2x_4\bar{x}_5x_6 + x_3x_5x_7x_9 + x_1x_3\bar{x}_6x_8 \\ & + \bar{x}_3\bar{x}_4\bar{x}_7\bar{x}_9 + \bar{x}_6x_7\bar{x}_8x_9 + x_2\bar{x}_4x_5x_7 + \bar{x}_2\bar{x}_3x_6\bar{x}_7 \\ & + x_2x_3x_7\bar{x}_9 + x_1x_7\bar{x}_8\bar{x}_9 + x_2x_3x_6x_9 + \bar{x}_3\bar{x}_7\bar{x}_8\bar{x}_9 \ . \end{aligned}$$

They terminated the double complement method after more than 6000 seconds of CPU time on a CDC 6600 computer without obtaining the prime implicants associated with this expression. Using a general factorization scheme, they were able to find the 87 prime implicants in about 400 CPU seconds.

This sum-of-products expression can be represented as a fault tree in the manner indicated by the introductory example of Subsection I.4.1 (Figure 3): Subscripts of expression variables are associated with basic nodes, a separate gate node with AND logic is created for each product, and the top node is an OR relation between these gate nodes. The MSDOWN method then essentially finds M^d_{top} by applying the dual algorithm to a family F of twenty sets, where each set is composed of variable indices in one of the above products, so $d\left[M^d_{top}\right]$ is found by two major applications of the dual algorithm. The FTAP program, implemented on a CDC 6400 computer (which is roughly comparable in speed to the CDC 6600), required less than 6 CPU seconds to find the 87 prime implicants.

I.4.4 Comments on the Choice of Method

The question naturally arises as to which of the three methods of this section is "best" for a particular modular subtree. When the subtree is small, say fewer than 20 gate nodes, these methods will not often differ widely in computational efficiency, and any of the algorithms is appropriate, unless the subtree has complementing arcs, in which case the Nelson method is usually preferable because it produces a prime implicant family. On the other hand, when the subtree is large, say more than 50 gate nodes, it is usually difficult to predict the relative efficiency of these methods. The analyst may have to rely on trial and error or previous computational experience with similar subtrees, combined with a few general considerations discussed here.

Let us first assume that the modular subtree contains no complementing arcs. MSDOWN or MSUP will most likely be selected in this case. MSDOWN is intended for use when the complete minimal implicant family M_i is required, and is apt to be more suitable for this purpose than the MSUP method, especially when a moderate or large number of replicated nodes are somewhat evenly distributed throughout the subtree. The general "downward" method, as well as the dual algorithm incorporated with MSDOWN, then offers some protection against the sudden appearance during processing of an unmanageable number of nonminimal sets. However, if the subtree contains a small number of replicated nodes, MSUP may be the faster of the two methods for finding the complete family, but the difference in efficiency will probably not be dramatic. MSUP, of course, is primarily intended for use in deriving a subfamily of M_i consisting of important or size restricted sets.

The Nelson method may seem superfluous for a subtree without complementing arcs, but when the complete minimal family M_i^d for the dual modular subtree has significantly fewer sets than the primal family M_i , this method is apt to surpass MSDOWN or MSUP. Of course, for the Nelson method to be successful, it is first essential that MSDOWN (or MSUP) be capable of finding M_i^d . One clue that suggests M_i^d might be small is a predominance of subtree gate nodes with OR logic. A more formal approach is to calculate rough upper bounds β_i and β_i^d , called subtree *binary indicated implicant counts*, on the number of sets in M_i and M_i^d ; a simple procedure for computing β_i and β_i^d is given below. For subtrees without complementing arcs, β_i and β_i^d are the same as counts of *binary indicated cut sets* and *binary indicated path sets* defined by Chatterjee [3], as long as the modular subtree is treated as an independent fault tree in the latter definitions, with largest simple modules for the subtree top event representing basic nodes. As a first approximation, it is usually reasonable to suppose that β_i^d exceeds the number of sets in M_i^d by one or maybe two orders of magnitude. When the complete family M_i^d can be generated, it may be feasible to find all of the implicants of $d\left[M_i^d\right]$ or perhaps only those satisfying an importance or size constraint. The Nelson method, with size or importance elimination enabled when applying the dual algorithm to M_i^d, can be considerably faster than the MSUP method for obtaining the desired subfamily.

Quantities β_u and β_{-u} are defined for a modular subtree with top node u ; for $i = +u$ or $i = -u$, $\beta_i^d \equiv \beta_{-i}$. For generality, we allow the subtree to contain complementing arcs. Suppose the MOCUS and MICSUP methods, as discussed in Subsection I.2.2, were modified to inhibit minimization. If then applied to the modular subtree, with largest simple modules for u treated like basic events, both methods would produce the same nonminimal family B_u or B_{-u} in terms of largest simple modules for u . Families B_u and B_{-u} are called *subtree binary indicated implicant families*. Fortunately, the number of sets in these families is easy to compute without deriving the families themselves; β_u and β_{-u} are these counts.

When the modular subtree involves no complementing arcs, M_u and M_{-u} are unique prime implicant families, with $M_u \subseteq B_u$ and $M_{-u} \subseteq B_{-u}$, so β_u and β_{-u} are upper bounds for the number of sets in M_u and M_{-u} . Moreover, in the absence of complementing arcs, the families M_u^d and M_{-u}^d associated with the dual subtree are also unique, and for $i = +u$ or $i = -u$, M_i^d can be obtained from M_{-i} by replacing each event j in an implicant of M_{-i} by its complementary event $-j$. Thus β_{-i} is also an upper bound on the size of M_i^d . On the other hand, when the modular subtree contains complementing arcs, the family M_i determined by algorithm MSDOWN or MSUP is generally not a prime implicant family for i , and it cannot be argued that $M_i \subseteq B_i$. However, for all practical purposes, it will very rarely be the case that the number of sets in M_i exceeds the number of sets in B_i .

Quantities β_u and β_{-u} are determined by the following rapid procedure: For each node v that is a largest simple module for the subtree top node u $\beta_v \leftarrow 1$ and $\beta_{-v} \leftarrow 1$. Consider nodes in G_u , the set of subtree nodes that are not largest simple modules, in upward order. For $v \in G_u$,

$$\beta_v \leftarrow \sum_{K \in \mathcal{D}(\ell_v, D_v)} \prod_{k \in K} \beta_k$$

$$\beta_{-v} \leftarrow \sum_{K \in \mathcal{D}(\# D_v - \ell_v + 1, D_{-v})} \prod_{k \in K} \beta_k .$$

The last values calculated are β_u and β_{-u} .

Finally, a few comments should be directed toward application of MSDOWN, MSUP, and Nelson methods to subtrees which contain complementing arcs. The workload for each algorithm is about the same as when complementing arcs are absent. However, the complete implicant family M_i produced by MSDOWN or MSUP is no longer guaranteed to be a prime implicant family for i . For some fault tree applications, this may be acceptable, or the analyst may wish to obtain M_i and from it generate a prime implicant family by any of a large variety of prime implicant algorithms available in the literature; for example, see [14]. Of course, implicant elimination based on size or importance should not be used in conjunction with MSUP to obtain a subfamily of M_i . A size or importance criterion, however, can be utilized to good advantage with the Nelson method. As remarked above, the quantity β_i^d ($\equiv \beta_{-i}$) can usually be assumed to exceed the number of sets in the family M_i^d produced by MSDOWN (or MSUP) for the dual subtree. So, as before if β_i^d is not too large, the Nelson method will probably be feasible.

PART II

USE OF THE FAULT TREE ANALYSIS PROGRAM

FTAP is a general purpose computer program for fault tree analysis employing the methodology of Sections I.3 and I.4. The bulk of the program consists of about 3500 FORTRAN statements, segmented into a driver routine and about 40 subroutines. Assembler code performs several simple operations that cannot be done in the context of standard FORTRAN. The FORTRAN portion of FTAP is compatible with nearly all FORTRAN compilers, but assembler routine packages are currently available only for CDC 6600/7600 and IBM 360/370 series machines. However, versions of these routines can easily be prepared in any assembler language according to specifications given in Section II.6.

Considerable effort has been expended to insure that FTAP will be easy to use. The input format is direct and unified, and input data is completely checked for correctness and consistency. Error messages are detailed, allowing the user to promptly identify problems involving program input or execution. Also, an ample number of comment cards are interspersed with the FORTRAN source statements to describe the code in terms of the algorithms of Part I; notation used for these comments is intended to resemble the notation scheme of Sections I.3 and I.4. Finally, FTAP has been extensively tested for reliable operation.

Sections II.1 through II.4 below describe program input and output; Section II.5 discusses the general procedure for implementing FTAP at a computer installation.

II.1 General Input Structure

The smallest logical units of input data are called *program instructions*, each of which is usually confined to a single 80-column punched card, though some instructions may be continued on additional cards. Program instructions are classified according to three major groups: *gate node definition*, *option*, and *execution.*

Gate node definition instructions specify a fault tree for analysis. These are always read first by the program, and if they are free of errors, a representation of the fault tree is stored in main memory. Errors in fault tree specification are messaged and cause processing to terminate.

One or more option instructions may follow fault tree specification, and information provided by these instructions is checked and stored. Options allow the user to (1) modify the fault tree, (2) select arbitrary gate nodes for which implicant families are to be found, (3) specify the methodology for obtaining implicant families, (4) enable implicant elimination on the basis of size or importance, and (5) control program printed and punched output.

The next card to be read after the option group is an execution instruction, which may be one of the two types we shall designate as **TREE* and **XEQ*. The *TREE instruction invokes an FTAP procedure that produces a structural description of the fault tree, essentially by listing modular subtrees and binary indicated implicant counts. The *XEQ instruction begins the operation of obtaining implicant families. Option instructions affect the processing initiated by an immediately following *TREE or *XEQ instruction, and this processing will be referred to as a *run*. Multiple runs are permitted; when a run is completed, the program reinitializes all memory locations except those associated with the input fault tree, so a group of options and an execution instruction for a new run may follow the execution instruction for the previous run. Options for a given run in no way affect other runs. The input data package for the FTAP program therefore has this general form:

```
fault tree specification
run  1  option instructions
run  1  *TREE or *XEQ instruction
run  2  option instructions
run  2  *TREE or *XEQ instruction
  .
  .
  .
run  n  option instructions
run  n  *TREE or *XEQ instruction.
```

For convenience, the same 80-column card format is used for all instructions and consists of eight fields across the width of the card. A particular instruction, however, will typically utilize only information punched in certain of these fields. Field 1 is composed of card columns 1-8. Fields 2 through 8 consist, respectively, of columns 11-18, 21-28, 31-38, 41-48, 51-58, 61-68, and 71-78.

The entry in field 1 is either a gate node name or an instruction name, left-justified in the field. FTAP automatically numbers fault tree nodes with positive integers in the scheme of Part I and allows the analyst the luxury of choosing names to replace these node numbers on program input and printed output. Node names may consist of any combination of eight or less characters. Instruction names are fixed strings of eight or less characters and are discussed in Section II.4.

Depending on the instruction, the entry in field 2 is either a positive integer, a decimal number in a FORTRAN E or F format, or one of the special characters plus (+) or asterisk (*). An entry may appear anywhere in field 2, except for an E-format decimal number, which must be right-justified.

Entries in fields 3 through 8 are again names of fault tree nodes, left-justified in these fields. The dash (-) may appear in any of the columns 20, 30, 40, 50, 60, or 70 if the field immediately to the right of the column contains a node name. Dashes represent event complementation.

II.2 Fault Tree Specification

The input fault tree is specified through a series of gate node definition instructions arranged in any order and followed by a card with the string "ENDTREE" left-justified in field 1. For a gate node u, the associated definition instruction provides the value ℓ_u of the logic indicator and the set D_u of immediate subevents. The first card of the instruction contains the node name in field 1 and names of immediate subnodes in fields 3 through 8. At least one subnode must appear, and no two fields may contain the same name. If node u is joined to an immediate subnode by a complementing arc, a dash should precede that particular subnode name.

The logic indicator value ℓ_u is a positive integer that may be placed anywhere in field 2; of course, ℓ_u may not exceed the number of immediate subevents. Optionally, either of the special characters plus or asterisk may be used in field 2, with a plus signifying a value of 1 for ℓ_u (an OR relation between subevent variables), and the asterisk signifying a value for ℓ_u equal to the total number of subevents (an AND relation).

When a node has more than six immediate subevents, additional subevent names may be entered in fields 3 through 8 on one or more cards which follow the first card and continue the gate node definition. Fields 1 and 2 on continuation cards are to be left blank. There is no restriction on the total number of immediate subevents.

As an example of fault tree specification, we consider again the tree of Figure 1, redrawn in Figure 5 with an unimaginative choice of node names. The tree is specified as follows (where each line is to be interpreted as a separate card):

```
Col. 1      11      21      31      41
↓           ↓       ↓       ↓       ↓
TOP         +       G2      G5      G6
G2          *       G3      G4
G3          +       G4      G5
G4          *       -G6     B11
G5          *       G7      G8
G6          *       B9      B10
G7          +       B12     B13
G8          +       B13     B14
ENDTREE
```

More than one fault tree can, in fact, be specified by a group of gate node definitions. For instance, if the instructions for TOP, G2, and G3 were deleted from the above list, FTAP would still accept the remaining instructions, though they represent two distinct trees with top nodes G4 and G5.

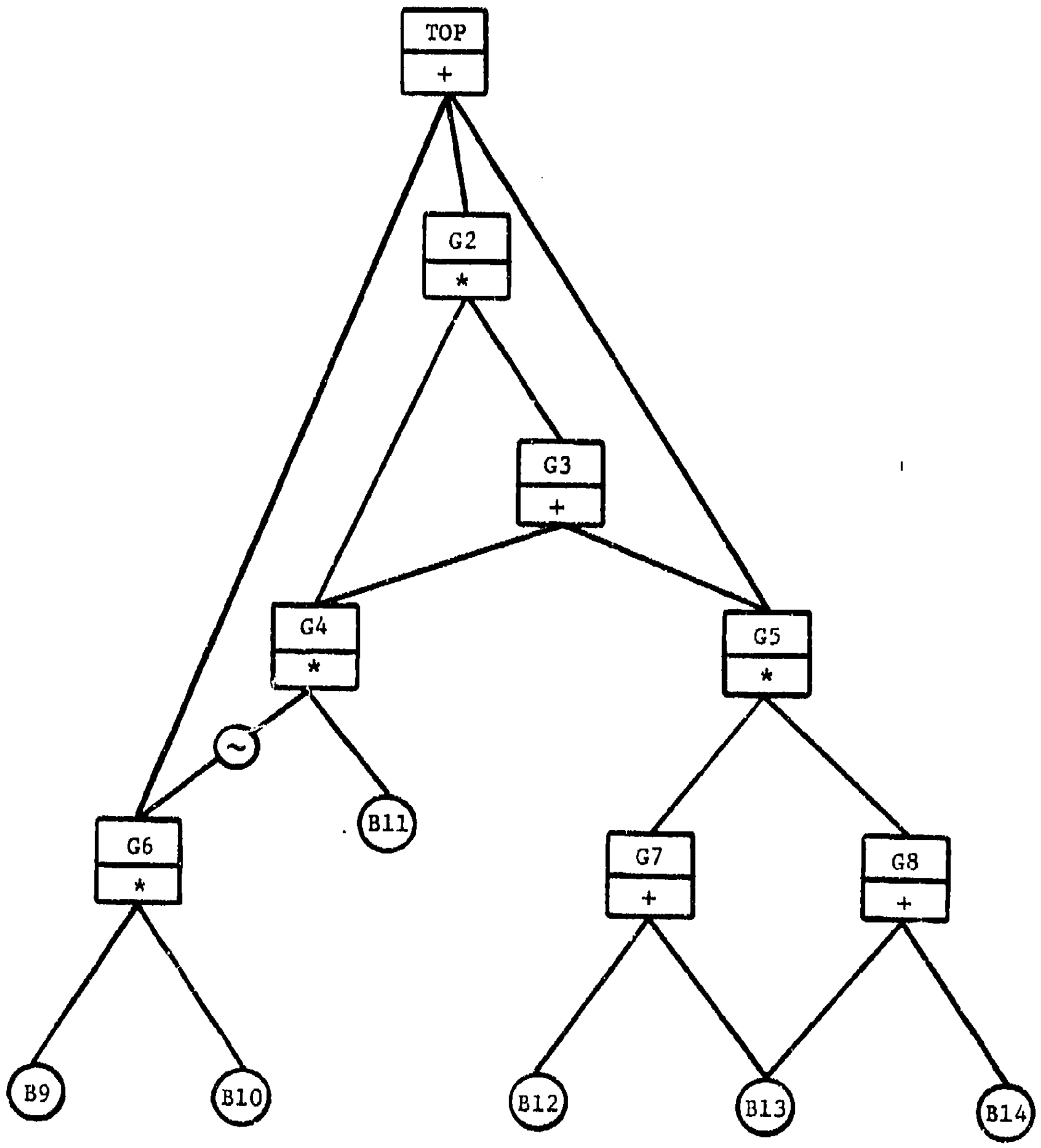


FIGURE 5

II.3 Execution Instructions

In some applications, the analyst may not wish to include any option instructions for a run; a *TREE or *XEQ instruction should then immediately succeed the ENDTREE card or the execution instruction for the previous run. An execution instruction consists simply of the name "*XEQ" or "*TREE" left-justified in field 1. In the absence of options, FTAP responds to a *XEQ instruction by seeking a minimal implicant family in terms of basic events for the fault tree top node. FTAP responds to a *TREE instruction by performing a structural analysis of the input tree and printing three types of information: (1) a representation of the tree, which is similar to a listing of gate node specification instructions; (2) an "inverse" tree, which identifies, for each gate or basic node u , the set of immediate supernodes of u ; and (3) a representation of each modular subtree whose top node is a simple module for the fault tree top node. Binary indicated implicant counts are also printed for each modular subtree and its dual.

As an illustration, assume that the ENDTREE card of the example tree specification is followed by the two instructions:

```
*TREE
*XEQ .
```

Output for the first run begins with the tree representation:

```
TREE FOR ANALYSIS

    (B) PRECEDES BASIC EVENTS

    NODE     LOGIC SUBEVENTS

    TOP        1          G2              G5            G6
    G2         2          G3              G4
    G3         1          G4              G5
    G4         2         -G6          (B) B11
    G5         2          G7              G8
    G8         1     (B)  B13         (B) B14
    G7         1     (B)  B12         (B) B13
    G6         2     (B)  B9          (B) B10
```

Next we obtain the "inverse tree":

```
INVERSE TREE

    NODE                IMMEDIATE SUPERNODES

    G2                     TOP
    G3                     G2
    G4                     G3       G2
    G5                     G3       TOP
    G6                     G4       TOP
    G7                     G5
    G8                     G5
    B10                    G6
    B11                    G4
    B12                    G7
    B13                    G7       G8
    B14                    G8
    B9                     G6
```

This is followed by modular subtree information, which completes the *TREE run:

```
MODULAR SUBTREES

 (B) PRECEDES BASIC EVENTS
 (M) PRECEDES LARGEST SIMPLE GATE MODULES FOR SUBTREE TOP NODE

 SUBTREE FOR NODE TOP

      TOP        1         G2        (M)  G5          (M)  G6
      G2         2         G3             G4
      G3         1         G4        (M)  G5
      G4         2    (M) -G6        (M)  B11

  SUBTREE BINARY INDICATED IMPLICANT COUNT  PRIMAL .4000E+01  DUAL .4000E+01

 SUBTREE FOR NODE G5

      G5         2         G7
      G7         1    (B)  B12       (B)  B13
      G8         1    (B)  B13       (B)  B14

  SUBTREE BINARY INDICATED IMPLICANT COUNT  PRIMAL .4000E+01  DUAL .2000E+01

 SUBTREE FOR NODE G6

      G6         2    (B)  B9        (B)  B10

  SUBTREE BINARY INDICATED IMPLICANT COUNT  PRIMAL .1000E+01  DUAL .2000E+01
```

The *XEQ run simply yields a minimal basic event family for the fault tree top node:

```
                    IMPLICANTS FOR EVENT  TOP

                       1         B13
                       2         B10        B9
                       3        -B10        B11
                       4        -B9         B11
                       5         B12        B14

                  CPU TIME FOR RUN   .349 SEC.
```

The flexibility of FTAP is due to a large variety of options discussed in the next section; even for simple fault trees, the analyst will probably wish to include some of these instructions before *TREE or *XEQ. All options affect a run initiated by a *XEQ instruction. However, when the *TREE instruction is used, the only options that are effective are those that modify the fault tree (TRUE, FALSE) or select gate events for analysis (PROCESS, ALL).

II.4 Option Instructions

The first card of each option instruction contains the option name in field 1. This initial card will be the only card for all instructions except TRUE, FALSE, PROCESS, NELSON, and IMPORT. Options TRUE, FALSE, PROCESS, and NELSON may be continued on subsequent cards in the same manner as gate node definition instructions, by leaving fields 1 and 2 blank on continuation cards; the number of continuation cards is not restricted. The IMPORT option usually consists of more than three cards but does not utilize the common continuation scheme. Options other than these also have fields 2 through 8 blank, except for MAXSIZE, which utilizes field 2.

Any option may be used for a run, but there are certain pairs of incompatible options, and use of both options is treated by FTAP as an error. In addition, a particular option may appear no more than once for a run. But no restriction is placed on the number of options that may be specified for a run or their input order.

We discuss options according to five functional categories.

II.4.1 Fault Tree Modification (TRUE, FALSE)

TRUE and FALSE permit any gate or basic event variables to be taken as identically true or false for a run. Node names are entered in fields 3 through 8 of these instructions, with preceding dashes signifying complementation; field 2 is blank.

The effect of setting event variables to true or false is accomplished by constructing a modified version of the input fault tree. An implicant family generated by FTAP in response to the *XEQ instruction then applies to this modified tree, as does tree structure data provided for the *TREE instruction. Nodes listed on TRUE and FALSE instructions do not appear in the modified tree, nor do any of their supernodes whose associated event variables become true or false. Some logic indicators for gate nodes in the new tree may, of course, differ from those in the input tree.

As an example, the Figure 5 tree is transformed to the tree of Figure 6 through the instruction:

```
Col. 1          21          31
↓               ↓           ↓
TRUE           -G3          B12
```

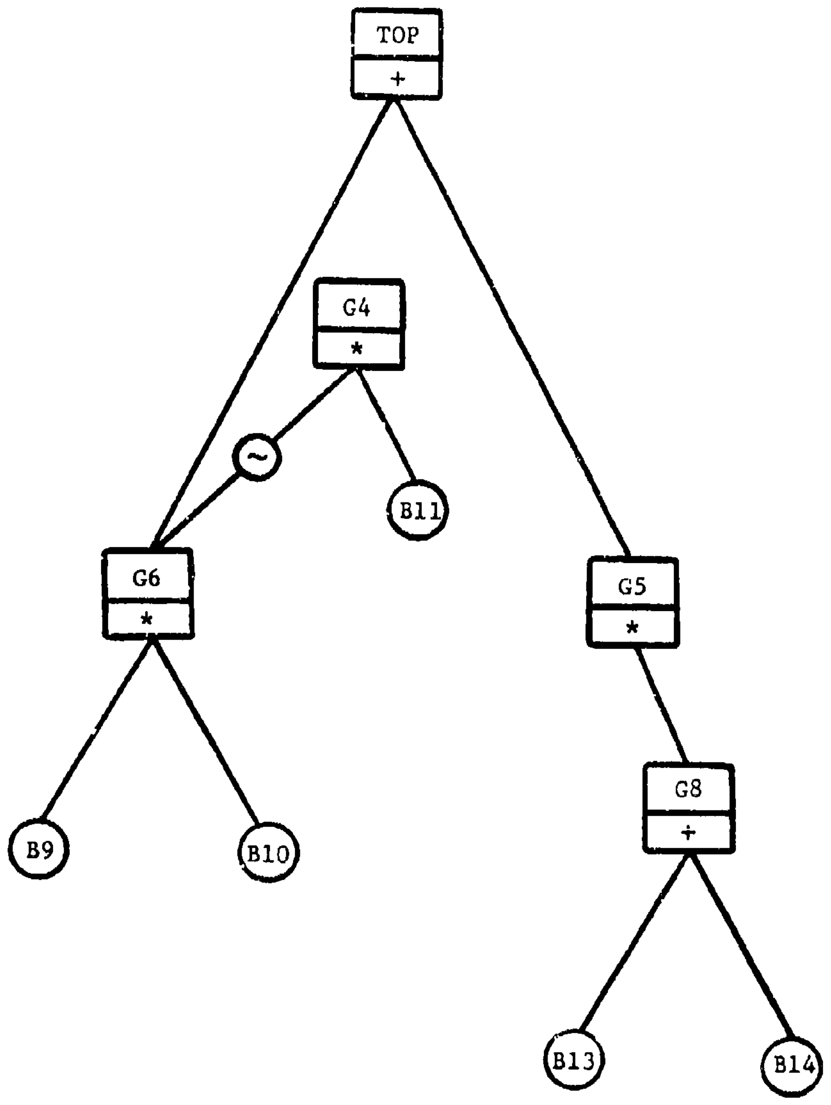


FIGURE 6

II.4.2 Gate Event Selection (PROCESS, ALL)

The analyst may wish to obtain implicant families for events associated with gate nodes other than the input tree top node; this is achieved by using a PROCESS or ALL instruction. The PROCESS instruction has gate node names in fields 3 through 8, with dashes optionally preceding these names. Field 2 is always blank. The ALL option consists simply of the string "ALL" in field 1. When the *XEQ instruction initiates the run, an implicant family is obtained for x_u if the name for node u appears without a preceding dash on a PROCESS instruction; a preceding dash results in an implicant family for x_{-u} $(\equiv \bar{x}_u)$. The PROCESS instruction, in fact, determines modular structure $\{M_j\}_{j \in M(Q)}$ by specifying the set of events Q . The ALL instruction is less selective, and, when used in conjunction with an *XEQ instruction, provides implicant families for x_u for every gate node u , as well as a family for x_{-u} if the fault tree contains complementing arcs. With the ALL option, the set Q for the modular structure is thus either the set G of all gate nodes or $G \cup (-G)$.

The ALL option is perhaps more useful when a *TREE instruction initiates the run. In this case, output from the tree structure analysis procedure includes information on the modular subtree for every gate node. On the other hand, the procedure provides modular subtree information only for nodes corresponding to events in $M(Q)$ if the set Q is selected through a PROCESS instruction.

When PROCESS and ALL options are absent, FTAP takes Q to consist only of the fault tree top node, unless the input fault tree has more than a single top node. In the latter case, absence of both instructions is treated as an error. Events in Q which are identically true or false are messaged at the beginning of run and excluded from further consideration.

PROCESS and ALL are incompatible, and it is an error to specify both for the same run.

II.4.3 Methodology Specification (PRIME, ALLNEL, NELSON, MSUP, MSDOWN, WRKFILES, MSONLY, DUAL, UPWARD, MINCHECK)

Options discussed here affect the manner in which FTAP obtains implicant families, so these instructions are only meaningful for runs initiated by the *XEQ instruction. Except for NELSON, these options consist only of the instruction name in field 1 of a card.

PRIME, ALLNEL, and NELSON instructions signal that the Nelson method is to be employed in obtaining certain minimal families M_j in the modular structure. PRIME indicates that this method is to be used only when the modular subtree for event j contains a complementing arc. PRIME thus guarantees that all families generated for a run consist of prime implicants; of course, this option has no effect if the input fault tree is devoid of complementing arcs.

The ALLNEL option, on the other hand, is effective for any input tree. In this case, the Nelson method is utilized in obtaining all M_j in the modular structure.

The NELSON option permits the Nelson method to be applied selectively for events corresponding to node names in fields 3 through 8 of this instruction. That is, if the name for some node u ccurs in one of these fields without a preceding dash, then as long as $u \in M(Q)$ M_u is determined by the Nelson method; a preceding dash has the same effect for M_{-u}. An event $j \notin M(Q)$ selected by this instruction is ignored, so the analyst should have some knowledge of $M(Q)$, perhaps derived from an earlier structural analysis of the fault tree.

If a modular subtree contains complementing arcs, it is possible for an event variable associated with the subtree top node to be identically true or false. The family M_j for a variable x_j which is identically false is always empty, and FTAP gives this result. However, if the variable x_j is identically true, this may not be apparent from FTAP results unless the Nelson method is utilized in finding M_j. The first task in this method is to find a complete minimal family M_j^d for the dual modular subtree, and M_j^d is empty if x_j is true. Should M_j^d be empty, FTAP prints an appropriate message and terminates the run.

PRIME, ALLNEL, and NELSON are incompatible with each other, and only one of the three may appear for a run.

Because FTAP automatically makes a reasonable choice between MSDOWN and MSUP algorithms in finding families M_j, the analyst will not often want to include a MSDOWN or MSUP option for a run. When the Nelson method is employed for a family M_j, FTAP automatically chooses the algorithm MSDOWN to first find a complete minimal family for the dual modular subtree, but the user may override this choice through the MSUP option. When the Nelson method is not employed, FTAP chooses the MSDOWN algorithm to find M_j unless implicant elimination based on size or importance is enabled, in which case the MSUP method is chosen. Again, either choice may be overridden through a MSUP or MSDOWN option. Since fields 2 through 8 of these options are blank, selective application to the families M_j is not possible. The presence of both options for the same run is treated as an error.

WRKFILES informs the program that sequentially organized file space on magnetic disk is available for use as working storage. FORTRAN file numbers 10, 11, and 12 must be assigned if this option is used. This storage is only available to subroutines that implement the dual algorithm. Though MSDOWN and ORDOWN methods both employ the dual algorithm, magnetic disk storage, when necessary, will most often be utilized in application of the Nelson method to large modular subtrees. Thus, the WRKFILES option will usually appear in conjunction with ALLNEL, PRIME, or NELSON. If the WRKFILES option has not been used for a run which must be terminated because of insufficient working space in main memory, a message may be printed suggesting that main memory could have been supplemented by magnetic disk storage. In this case, it is reasonable for the analyst to try a rerun with a WRKFILES instruction.

Once the modular structure has been found, the procedure for finding implicant families in terms of basic events is very efficient computationally, but for large fault trees, this final step might require a great deal of main memory workspace. The MSONLY option instructs FTAP to bypass this step, so only the modular structure is determined. This instruction will also lead to more efficient use of main memory in obtaining the modular structure, since families M_j need not be retained once they have been printed. Also, FTAP includes a subroutine which, when provided with the family M_j in terms of largest simple modules for j, counts the number of implicants in the minimal basic event family I_j (without deriving this family). Separate counts are accumulated by implicant size and printed by the subroutine. If MSONLY is specified, the routine is called for each $j \varepsilon M(Q)$.

The DUAL option simply indicates that all implicant families for a run are to be derived for the dual of the input fault tree. Thus, if an implicant family associated with the primal tree consists of system cut sets, a corresponding minimal path set family is obtained by using the DUAL instruction.

The UPWARD option invokes an algorithm not explicitly stated in Part I. This method closely resembles the MSUP algorithm: The general MSUP technique is applied to the entire fault tree rather than a modular subtree, with basic events replacing largest simple modules. Thus implicant families are generated directly in terms of basic events without utilizing the modular structure. The UPWARD option may be useful when the required minimal implicant families in terms of basic events are expected to be small, which might be the case even for large fault trees if size and importance elimination options are included for a run. Because the modular structure is not determined when this option is specified, certain other options are incompatible with UPWARD. These are NELSON, PRIME, MSDOWN, MSUP, MSONLY, MODSIZE, MSPRINT, and MSPUNCH, the last three of which we will consider shortly.

The MINCHECK instruction is only effective when it accompanies the UPWARD option. MINCHECK specifies that minimization only be applied to implicant families for events in the set Q determined by PROCESS or ALL options, or in the absence of these options, to the family for the fault tree top node. Thus, intermediate families generated for events that are not of interest to the analyst are not minimized.

II.4.4 Control of Printed and Punched Output (MSPRINT, STATUS, DSTATUS, PUNCH, MSPUNCH, NOPRINT)

These options control output information regarding implicant families and are effective only for runs initiated with a *XEQ instruction.

MSPRINT instructs FTAP to include the modular structure families in printed output. This option is unnecessary when MSONLY is provided, because MSONLY also enables printing of the modular structure. As an illustration, suppose the ENDTREE card for specification of the Figure 5 tree is followed by the instructions:

```
MSPRINT
*XEQ .
```

Modular structure output is then:

```
IMPLICANTS IN TERMS OF LARGEST SIMPLE MODULES

     IMPLICANTS FOR EVENT  G6

         1         B10        B9

     IMPLICANTS FOR EVENT -G6

         1        -B9
         2        -B10

     IMPLICANTS FOR EVENT  G5

         1         B12        B14
         2         B13

     IMPLICANTS FOR EVENT  TOP

         1         G5
         2         G6
         3        -G6         B11
```

The STATUS option yields information on the progress of generating each minimal implicant family, giving the number and maximum size of implicants in various intermediate families, as well as data on computation times and the amount of unused main memory. STATUS provides a brief record of each iteration of the MSDOWN method. As an illustration, consider the instruction group:

```
MSPRINT

STATUS

*XEQ .
```

A sample of the output for the event TOP modular structure family from a run initiated by these instructions is as follows:

```
   LARGEST SIMPLE MODULES FOR TOP
       G5         B11       G6

   --------------------   EVENT  TOP    DOWNWARD   --------------------

   NUMBER OF IMPLICANTS IN TABLE      3             MAXIMUM LENGTH      1
   NUMBER OF IMPLICANTS IN TABLE      3             MAXIMUM LENGTH      2
   NUMBER OF IMPLICANTS IN TABLE      3             MAXIMUM LENGTH      1
   MINIMIZATION
                                      3             MAXIMUM LENGTH      1
   NUMBER OF IMPLICANTS IN TABLE      3             MAXIMUM LENGTH      2
   MINIMIZATION
                                      3             MAXIMUM LENGTH      2
   UNUSED STORAGE BEGINS AT       361    CPU TIME FOR EVENT       .070 SEC

IMPLICANTS FOR EVENT  TOP

    1         G6
    2         G5
    3        -G6        B11
```

STATUS information for the Nelson method is similar to the above, except the various "downward" lines refer to implicants for the dual subtree, and data on the number and size of implicants obtained from the dual algorithm precedes storage and time data.

Information for the MSUP method is somewhat different. The run instructions

```
MSPRINT

MSUP

STATUS

*XEQ
```

give this output for the event TOP modular structure family:

```
   LARGEST SIMPLE MODULES FOR TOP
       G5         B11        G6

   --------------------  EVENT  TOP      DOWNWARD    --------------------

   NUMBER OF IMPLICANTS IN TABLE       3              MAXIMUM LENGTH      1
   MINIMIZATION                        3              MAXIMUM LENGTH      1

   --------------------  EVENT  G2       DOWNWARD    --------------------

   NUMBER OF IMPLICANTS IN TABLE       1              MAXIMUM LENGTH      1

   --------------------  EVENT  G4       DOWNWARD    --------------------

   NUMBER OF IMPLICANTS IN TABLE       1              MAXIMUM LENGTH      2

   --------------------  EVENT  G4        UPWARD     --------------------

   NUMBER OF IMPLICANTS IN TABLE       1              MAXIMUM LENGTH      2

   --------------------  EVENT  G2        UPWARD     --------------------

   NUMBER OF IMPLICANTS IN TABLE       1              MAXIMUM LENGTH      2

   --------------------  EVENT  TOP       UPWARD     --------------------

   NUMBER OF IMPLICANTS IN TABLE       3              MAXIMUM LENGTH      2
   MINIMIZATION                        3              MAXIMUM LENGTH      2

   UNUSED STORAGE BEGINS AT      387    CPU TIME FOR EVENT      .129 SEC

IMPLICANTS FOR EVENT  TOP

    1          G6
    2          G5
    3          -G6          B11
```

The "downward" information now represents successive applications of the ORDOWN algorithm to events TOP, G2, and G4. These events are then considered in upward order, as the family for TOP in terms of largest simple modules is generated. The "upward" information format is also used in a rather obvious way to chart the progress of constructing basic event families, whether this construction proceeds from the modular structure or in the manner associated with the UPWARD option.

DSTATUS causes the subroutine package for the dual algorithm to provide data on the sizes of various tables associated with that algorithm. This data is only printed when the Nelson method is used and the subroutine package is applied to the implicant family for a dual modular subtree. Some familiarity with Reference [17] is required to interpret this output.

PUNCH and MSPUNCH options allow implicant families to be punched on 80-column cards for input to other programs. FORTRAN file number 7 should be assigned to the card punch (or magnetic disk) if these instructions are used. Events associated with the input fault tree are represented by positive and negative integers [illegible] punched output, and whenever the MSPUNCH option is used, this numbering scheme is the same as suggested in Part I. MSPUNCH enables punching of the modular structure, and determines that, for the q nodes of the input fault tree, integers 1 to p are to represent gate nodes on punched output and $p + 1$ to q are to represent basic nodes. The PUNCH option causes basic event families to be punched. Unless MSPUNCH accompanies the PUNCH instruction for a run, integers 1 to $q - p$ number basic nodes on output.

When either MSPUNCH or PUNCH is used, the first group of punched cards for a run gives the correspondence between node names and numbers. The initial card of the group has the FORTRAN format (5HNAMES,I5), where the single integer field contains the number of names (which will be q if MSPUNCH is specified and $q - p$ if only PUNCH is specified). On the remaining cards node numbers are paired with node names, with up to five pairs appearing on a card in the format (5(I5,3H - ,A8)).

The modular structure, if requested, is given by the next group of cards, whose initia card has a (5HIMPMS,I5) format, containing the run number in the integer field. The representation of each family M_j then begins with a (5HEVENT,I5,I6) format header card, having the positive or negative integer j in the first field and the number of implicants in the family in the second. Following the header card, each implicant of the family starts on a separate card with a (16I5) format and may continue on additional cards with the same format. On the first card for an implicant, field 1 always contains the number of events in the implicant. These events are represented in fields 2 through 16 of the first card and 1 through 16 on subsequent cards.

Output for basic event families is preceded by a (5HIMPBE,I5) format card, with the run number in the integer field. The general format for representing these families follows that of the modular structure, where the basic event families $\{I_j\}_{j\varepsilon Q}$ take the place of $\{M_j\}_{j\varepsilon M(Q)}$. Again, Q either contains the fault tree top node or events indicated by a PROCESS or ALL instruction.

Finally, the analyst may sometimes wish to obtain punched output but suppress printed output for large basic event families. In such cases, the NOPRINT option should accompany the PUNCH option. NOPRINT only suppresses printing of basic event families and does not affect the MSPRINT option.

II.4.5 Implicant Elimination Based on Size and Importance (MAXSIZE, MODSIZE, IMPORT)

These options are compatible with all of the algorithms MSDOWN, MSUP, and Nelson, as well as the method associated with the UPWARD instruction. If size or importance options are included for a run and options MSDOWN, PRIME, ALLNEL, and NELSON are absent, the MSUP algorithm is chosen automatically by FTAP, even for modular subtrees containing complementing arcs. However, it has been pointed out in Part I that when the MSUP method is applied to a subtree with complementing arcs, the resulting subfamily of size or importance restricted sets may not be meaningful. Thus, in utilizing these options, the analyst will usually want to insure that the Nelson method is employed for such subtrees.

The MAXSIZE option imposes a uniform size restriction on implicants in the modular structure and basic event families generated by FTAP. Field 2 of this instruction gives the maximum number of events permitted in an implicant; a positive integer may appear anywhere in the field. Fields 3 through 8 of the card are blank.

As discussed in Part I, modular size importance is often a more efficient criterion for implicant elimination than a simple size restriction. This criterion is applied in the manner suggested in Subsection I.3.2, which we briefly recall: The subfamilies $\{M_j'\}_{j\varepsilon M(Q)}$ are generated in "upward" order, with M_j constructed following families for subevents of j . An implicant M is retained in M_j' only if $\sum_{m\varepsilon M} \sigma(m)$ does not exceed the fixed size restriction, where $\sigma(m) = 1$ if m is a basic event, and for m a gate event, $\sigma(m)$ is available from an earlier computation which followed construction of M'_m ,

$$\min_{K \epsilon M'_m} \left(\sum_{k \epsilon K} \sigma(k) \right) \qquad (M'_m \neq \emptyset) \ .$$

FTAP implements elimination based on modular size importance when the MODSIZE option accompanies MAXSIZE; in this case, MAXSIZE specifies the fixed size restriction. MODSIZE is not effective in the absence of the MAXSIZE option.

FTAP also allows for implicant elimination based on the product importance criterion. Here an implicant M is retained in M'_j only if $\prod_{m \epsilon M} \iota(m)$ exceeds some critical value c , where $\iota(m)$ is an arbitrary value between 0 and 1 for m a basic event, and for m a gate event, $\iota(m)$ has been determined from

$$\max_{K \epsilon M'_m} \left(\prod_{k \epsilon K} \iota(k) \right) \qquad (M'_m \neq \emptyset) \ .$$

The criterion is applied again when basic event families are obtained from the modular structure.

The product importance option requires a group of cards to specify the values $\iota(\cdot)$ for basic events and the critical value c . The first card of the group contains only the option name "IMPORT" in field 1; other fields are blank. Cards which assign $\iota(\cdot)$ values follow this initial card. These cards always have field 1 blank, a positive decimal value between 0 and 1 in field 2, and basic event names in fields 3 through 8, with optional dashes preceding these latter fields. The value in field 2 may be in FORTRAN E or F format and must contain a decimal point. An F-format item, such as .5 or .001, may appear anywhere within the field, but E-format items, such as 1.25E-2 or .1E-1, must be right-justified. Should the name for basic node u appear on a card, $\iota(u)$ is assigned the value in field 2 of that card when the name is not preceded by a dash; a preceding dash causes the field 2 value to be assigned to $\iota(-u)$. As many cards as desired may be used to set $\iota(\cdot)$ values, but it is not required that values be provided for all basic events: FTAP assigns $\iota(k)$ a default of 1 for any event k not represented on one of these cards.

The card that must terminate the product importance group has the string "LIMIT" left-justified in field 1 and a decimal value between 0 and 1 again in field 2. Fields 3 through 8 are blank. The field 2 value is the critical value c for the importance test.

As a simple illustration of the above options, suppose for the example tree, the analyst desires only prime implicants consisting of a single basic event, and implicants involving node B9 are not of interest. Suitable cards for this run are:

```
Col. 1         11          21          31
↓              ↓           |           |
MAXSIZE        1           |           |
IMPORT                     ↓           ↓
               .4          B9         -B9
LIMIT          .5
PRIME
*XEQ
```

II.5 Program Implementation

FTAP is available in two distinct versions that differ only in the internal storage format for representing implicant sets. FTAP1 stores an implicant as a variable number of consecutive computer words in main memory. The first word of the group contains the integer number of events in the set, and a positive or negative integer value in each of the remaining words identifies an implicant event. FTAP2, on the other hand, stores an implicant set as a fixed group of consecutive words, and a fault tree event is associated with a unique bit position in one of these words. Bit positions corresponding to events in the implicant contain the value 1, whereas other bit positions contain 0. The fixed number of words required for an implicant set depends on the computer word length, the particular stage of FTAP processing, and whether the input fault tree contains complementing arcs. When modular structure families are constructed for a fault tree without complementing arcs, the number of words must be sufficient to accommodate one bit position for each gate and basic node. Fur fault trees with complementing arcs, two bits are needed for a node, one for each event associated with the node. When basic event families are constructed, the situation differs only on the fact that bit positions are not needed for representation of gate events.

FTAP2 is the more efficient of these two versions in terms of computation time and should be chosen for most applications. However, for large fault trees (having, say, more than 200 gate nodes), it is often feasible only to obtain implicants having a small number of events. The storage format of FTAP1 becomes an advantage in such applications.

FTAP1 and FTAP2 are designed for use on most general purpose computers. The codes have been carefully prepared to ensure that program logic is tight and efficient, and subroutines for minor tasks such as sorting and searching use good standard algorithms, as given in [9]. The FORTRAN portion of each program conforms to ANSI specifications, except for array subscripts, which are apt to consist of expressions using two or more simple FORTRAN integer variables with addition, subtraction, and multiplication operations, and sometimes the integer absolute value operation. Most FORTRAN compilers allow such expressions.

Main memory work space for either code is confined to one single subscripted integer array, denoted by the FORTRAN name IA. Storage in this array is dynamically allocated for maximum efficiency in use of main memory. Because fault trees of appoximately the same size may differ considerably in their structure, it is difficult to state even roughly how large IA should be to accommodate analysis of a fault tree with some given number of nodes. The analyst should make IA as large as feasible for the environment in which the program is implemented; for instance, if the program is required to execute in a fixed partition of computer main memory, then the object code length plus storage for IA should fill the partition. If the environment is such that program use becomes more inconvenient as storage requirements increase, an initial length of IA should be chosen perhaps between 300 and 1000 times the maximum number of gate nodes in any tree to be analyzed; this length may then be increased as necessary.

Implementation of FTAP1 or FTAP2 is accomplished according to the following steps:

1. The desired dimension of the array IA should be set in the declarative statement for this array near the beginning of the main program. Since the code must be capable of determining when storage requirements exceed availability, the length of the array must be provided for internal program use. This is done by initializing the variable IASIZE through a FORTRAN DATA statement, which also appears near the beginning of the main program.
2. The first executable statement in the main program for FTAP2 assigns a positive integer value to the variable LWORD. This value should be set to the length of a computer word less one.
3. When accessed by other routines, the REAL function TIME returns the amount of elapsed time since the beginning of the computer job. The proper form of the subroutine CALL statement in function TIME may depend on the particular computer installation, and this statement should be modified accordingly.
4. The group of assembler language routines should be chosen to correspond to both the computer and FORTRAN compiler used. Three groups of assembler language routines are supplied with FTAP1 or FTAP2: for use with (1) CDC 6600/7600 machines and RUN compiler linkage convention, (2) CDC 6600/7600 machines and FTN compiler linkage convention, or (3) IBM 360/370 machines (G or H compiler linkage convention). To implement FTAP1 or FTAP2 on other machines, one or more assembler routines must be prepared according to specifications given in the following section.

II.6 Specifications for Assembler Routines

The routines discussed in this section are all very simple, and the largest should not require many more than 25 statements in any assembler language. FTAP1 utilizes only routine CCM, but all routines are accessed by FTAP2.

1. CCM (IW1, IW2, ITEST):

CCM logically compares the contents of computer words IW1 and IW2 and returns the result of the comparison in word ITEST. If contents of IW1 and IW2 are identical, then ITEST will contain a 0 ; otherwise the value in ITEST depends on the highest bit in which the words differ. When this bit is 1 in IW1 and 0 in IW2, ITEST is returned as 1 ; in the reverse situation ITEST is returned as -1 .

2. ORM (IW1, IW2, IOR):

The contents of word IOR returned by this routine is simply a logical OR of words IW1 and IW2.

3. ANDM (IW1, IW2, IAND):

ANDM returns in word IAND the result of a logical AND of IW1 and IW2.

4. PUTM (LV, IV, IW):

When PUTM is accessed, IV is an array of successive words containing positive integer values in increasing order. No value exceeds the number of bits in a computer word. The location LV contains a positive integer representing the length of IV. The function of PTUM is to place a 1 in each bit position of word IW numbered by an integer in array IV; other bit positions are set to 0 . As an example, suppose the computer word length is 16, and PUTM is accessed with 4 in LV, and IV(1) through IV(4) contain, respectively, 2, 5, 7, and 16. On return, IW then contains the bit pattern "1000000001010010." The bit numbering for this example is increasing from right-to-left.

5. GETM (IW, LV, IV):

GETM performs the reverse operation of PUTM. On return, IV is a vector of consecutive words containing bit numbers for all bits that are 1 in word IW. These integers are in increasing order in IV, and the number of integers in this vector is returned in LV. GETM may be accessed with IW having 0's in all bit positions, in which case LV is returned with an integer value of 0 .

6. BCM (IW, NBITON):

BCM returns in NBITON the count of bit positions containing 1 in word IW. The value in NBITON is thus an integer between 0 and the number of bits in a computer word.

REFERENCES

[1] Aho, A., J. Hopcroft and J. Ullman, THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS, Addison-Wesley, Reading, Mass., 1974.

[2] Barlow, R. E. and F. Proschan, STATISTICAL THEORY OF RELIABILITY AND LIFE TESTING, Holt, Rinehart and Winston, New York, 1975.

[3] Chatterjee, P., "Fault Tree Analysis: Reliability Theory and Systems Safety Analysis," ORC 74-34, Operations Research Center, University of California, Berkeley, (1974).

[4] Chatterjee, P., "Modularization of Fault Trees: A Method to Reduce the Cost of Analysis," in RELIABILITY AND FAULT TREE ANALYSIS, R. E. Barlow, J. B. Fussell and N. D. Singpurwalla, (editors), SIAM, 1975.

[5] Edmonds, J. and D. R. Fulkerson, "Bottleneck Extrema," Journal of Combinatorial Theory, Vol. 8, pp. 299-306, (1970).

[6] Fussell, J. B. and W. E. Vesely, "A New Methodology for Obtaining Cut Sets," American Nuclear Society Transactions, Vol. 15, No. 1, pp. 262-263, (June 1972).

[7] Fussell, J. B. et al., MOCUS: A COMPUTER PROGRAM TO OBTAIN MINIMAL SETS, Aerojet Nuclear Co., Idaho Falls, 1974.

[8] Hulme, B. L. and R. B. Worrell, "A Prime Implicant Algorithm with Factoring," (prepublication copy).

[9] Knuth, D. E., THE ART OF COMPUTER PROGRAMMING, VOL. 3, SORTING AND SEARCHING, Addison-Wesley, Reading, Mass., 1970.

[10] Lambert, H. E., "Fault Trees for Decision Making in Systems Analysis," Report UCRL-51829, Lawrence Livermore Laboratories, Livermore, California, (1975).

[11] Nelson, R. J., "Simplest Normal Truth Functions," Journal of Symbolic Logic, Vol. 20, pp. 105-108, (1955).

[12] Pande, P. K., M. E. Spector and P. Chatterjee, "Computerized Fault Tree Analysis: TREEL and MICSUP," ORC 75-3, Operations Research Center, University of California, Berkeley, California, (1975).

[13] Pelto, P. and W. Purcell, "MFAULT: A Computer Prcgram for Analysizing Fault Trees," Report BNWL-2145, Battelle Pacific Northwest Laboratories, Richland, Washington, (1977).

[14] Reusch, B., "On the Generation of Prime Implicants," Report TR 76-266, Department of Computer Science, Cornell University, Ithaca, New York, (1976).

[15] Rosenthal, A., "A Computer Scientist Looks at Reliability Analysis," in RELIABILITY AND FAULT TREE ANALYSIS, R. E. Barlow, J. B. Fussell and N. D. Singpurwalla, (editors), SIAM, 1975.

[16] Whitesitt, J., BOOLEAN ALGEBRA AND ITS APPLICATIONS, Addison-Wesely, Reading, Mass., 1961.

[17] Willie, R., "A Computer Oriented Method to Find Boolean Duals," (forthcoming).

[18] Worrell, R. B., "Using the Set Equation Transformation System in Fault Tree Analysis," in RELIABILITY AND FAULT TREE ANALYSIS, R. E. Barlow, J. B. Fussell and N. D. Singpurwalla, (editors), SIAM, 1975.